A MARSHALL CAVENDISH 13 COMPUTER COURSE IN WEEKLY PARTS

INPUT

LEARN PROGRAMMING - FOR FUN AND THE FUTURE



UK £1.00 Republic of Ireland £1.25 Malta 85c Australia $2.25 New Zealand $2.95

INPUT

Vol 1 No 13

GAMES PROGRAMMING 13

COMPLETING THE ADVENTURE 385

Fill in the fine details that give an adventure the qualities that make it special

BASIC PROGRAMMING 29

GETTING THINGS IN ORDER 392

Three different methods for sorting—whether alphabetically or numerically

PERIPHERALS

THE SPEAKING COMPUTER 398

A look at speech synthesizers, the hardware that gives every computer its say

MACHINE CODE 14

A COMMODORE ASSEMBLER PROGRAM 402

Program your Commodore 64 to take the hard work out of converting assembly language into machine code

BASIC PROGRAMMING 30

THE POWERS THAT BE 406

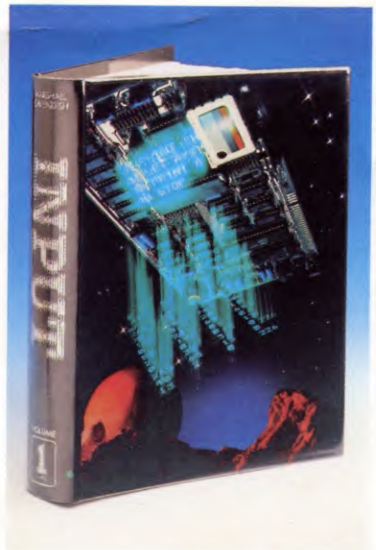
The power function—often overlooked, but a tool with many applications

INDEX

The last part of INPUT, Part 52, will contain a complete, cross-referenced index. For easy access to your growing collection, a cumulative index to the contents of each issue is contained on the inside back cover.

PICTURE CREDITS
Front cover, Jeremy Gower. Pages 385, 386, 389, 390, Neil Winstanley. Page 392, Tudor Art Studios. Pages 394, 396, Graeme Harris. Page 398, BBC Picture Library. Page 399, Jeremy Gower. Page 401, Tony Lodge. Pages 403, 404, Paul Chave. Pages 406, 408, 410, Digital Arts.

Published by Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA, England. Printed by Artisan Presss, Leicester and Howard Hunt Litho, London.



There are four binders each holding 13 issues.

QUERIES: When writing in, please give the make and model of your computer, as well as the Part No., page and line where the program is rejected or where it does not work. We can only answer specific queries – and please do not telephone. Send your queries to INPUT Queries, Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA.

INPUT IS SPECIALLY DESIGNED FOR:

The SINCLAIR ZX SPECTRUM (16K, 48K, 128 and +), COMMODORE 64 and 128, ACORN ELECTRON, BBC B and B+, and the DRAGON 32 and 64.

In addition, many of the programs and explanations are also suitable for the SINCLAIR ZX81, COMMODORE VIC 20, and TANDY COLOUR COMPUTER in 32K with extended BASIC. Programs and text which are specifically for particular machines are indicated by the following symbols:

SPECTRUM 16K, 48K, 128, and +
COMMODORE 64 and 128
ACORN ELECTRON, BBC B and B+
DRAGON 32 and 64
ZX81
VIC 20
TANDY TRS80 COLOUR COMPUTER

COMPLETING THE ADVENTURE

- A HELP ROUTINE
- DEALING WITH THE INSPECTOR
- TROUBLE WITH THE BRICK
- LIGHTING THE LAMP
- INSTRUCTIONS



The *INPUT* adventure is almost complete. All that remains are the routines which make the adventure unique—the special routines which give the game its character.

With most of the major elements of your adventure game filled in, now's the time to add the finer details of the program. There are still things like hazards and warnings which have yet to be incorporated—and you still have not provided a way out after successful completion of the adventure. You also have to add the instructions on how to play the game.

Because many of these routines are written to contain details which are specific to one adventure only, the programming which follows will complete the *INPUT* adventure game, and show you in general terms what is involved in taking the program to this stage. You will not be able to use these routines without modification in another adventure. But in the following article you will see how you can adapt the principles to suit your own, original ideas.

YOU NEED HELP

The adventurer will probably need some help if you've written a good adventure game. So how about some (helpful?) hints?

These will take the form of messages PRINTed out by the computer in response to the players request for HELP. What the messages are, and where they are available, is at the discretion of you, the programmer. You can choose, if you wish, to have no messages at all, or to make them deliberately misleading, or to provide help only in a few isolated locations. To see what it would be worth including, the first step is to go back to your original plan for the adventure storyline.

In the *INPUT* adventure, there are several points where it might be worth putting in a short message to the player. For example, you might consider giving a warning about the darkened room, so that if a player is in an adjacent location, a request for HELP might

provoke a message like: LOOK BEFORE YOU LEAP, or something even more cryptic.

Another place where you might want to provide a warning is at the river bank where, conditional on carrying the brick, the adventurer may be at risk of drowning, should he or she decide to swim.

Of course, you could go on and on looking for places where you might want to provide help, but let's suppose that you decide not to be too helpful, and only to provide a message in one place, the river. You need to refer this to its location number, number 7, and to the variable which records the presence of the brick, OB(2):

```
3100 REM **HELP**
3110 IF L<>7 OR B(2)<>-1 THEN
   PRINT "SORRY, I CAN'T HELP YOU
   HERE!": GOTO 330
3120 PRINT "BRICKS WEIGH A LOT AND
   IT'S□□□□□MAKING YOUR ARM
   ACHE": GOTO 330
```

```
3100 REM ** HELP **
3110 IF L<>7 OR OB(2)<>-1 THEN
   PRINT "SORRY, I CAN'T HELP YOU
   HERE!":GOTO 330
3120 PRINT "BRICKS WEIGH A LOT AND IT'S
   MAKING YOUR□ARM ACHE":
   GOTO 330
```

```
3100 REM **HELP**
3110 IF L<>7 OR OB(2)<>-1 THEN
   PRINT"SORRY, I CAN'T HELP YOU
   HERE!":GOTO 330
3120 PRINT"BRICKS WEIGH A LOT AND
   IT'S□□□□□MAKING YOUR ARM
   ACHE":GOTO330
```

If the adventurer isn't at the river (L<>7), or is not carrying the brick (OB(2)<>-1 or B(2)<>-1), Line 3110 PRINTs the message SORRY, I CAN'T HELP YOU HERE! If the brick is being carried on arrival at the river and help is requested, then the warning BRICKS WEIGH A LOT AND IT'S MAKING YOUR ARM ACHE appears, PRINTed by Line 3120.

There's nothing to stop you adding a whole list of conditions and related help messages if you want to.

THE TAX INSPECTOR

The adventure has a marauding baddie in the form of a tax inspector. The tax inspector's aim is to try to recover some of the adventurer's unpaid taxes by confiscating one of the objects being carried. He's not very particular about what he gets, though, because sometimes he'll even take a brick in payment!

If the adventurer isn't lucky enough to be carrying anything when the tax inspector calls, he'll be locked in a dungeon to rot for evermore. The game then ends.

The role of the tax inspector is to provide an element of chance, appearing unpredictably, regardless of location or other conditions. Like other examples of chance in programming, you can do this by using the RND function. Otherwise, he can be treated much like one of the objects, except that his location is set randomly, not fixed.

Here are the extra lines you'll need to make the tax inspector appear:

```
320 IF RND<(1/15) AND TA=0
   THEN LET B(7)=L:
   LET TA=1
480 IF B(7)=L AND I<>10 THEN
   GOTO 1590
```

```
320 IF INT(RND(1)*15+1)=1 AND TA=0
   THEN OB(7)=L:TA=1
480 IF OB(7)=L AND I<>10 THEN 1590
```

```
320 IF RND(15)=1 AND TA=0 THEN
   OB(7)=L:TA=1
480 IF OB(7)=L AND I<>10
   THEN1590
```

```
320 IFRND(15)=1 AND TA=0
   THENOB(7)=L:TA=1
480 IF OB(7)=L AND I<>10 THEN 1590
```

The Acorn and Spectrum have set all these variables to zero initially.

Line 320 in all of the programs gives the adventurer a 1 in 15 chance of meeting the inspector—the purpose of the 15 in the RND part of the line. He's only allowed to appear once during the game, so you need a variable—TA—to indicate if he has, or not.

If the random number is 1 (or less than a fifteenth on the Spectrum) and the inspector hasn't yet appeared, then Line 320 adjusts the value in the object location array corresponding to the tax inspector. The tax inspector message is displayed as if he were an object—it is stored in the long description array.

Line 480 is concerned with killing the tax inspector. It simply checks if you have tried to kill him. If you haven't, the program jumps to Line 1590.

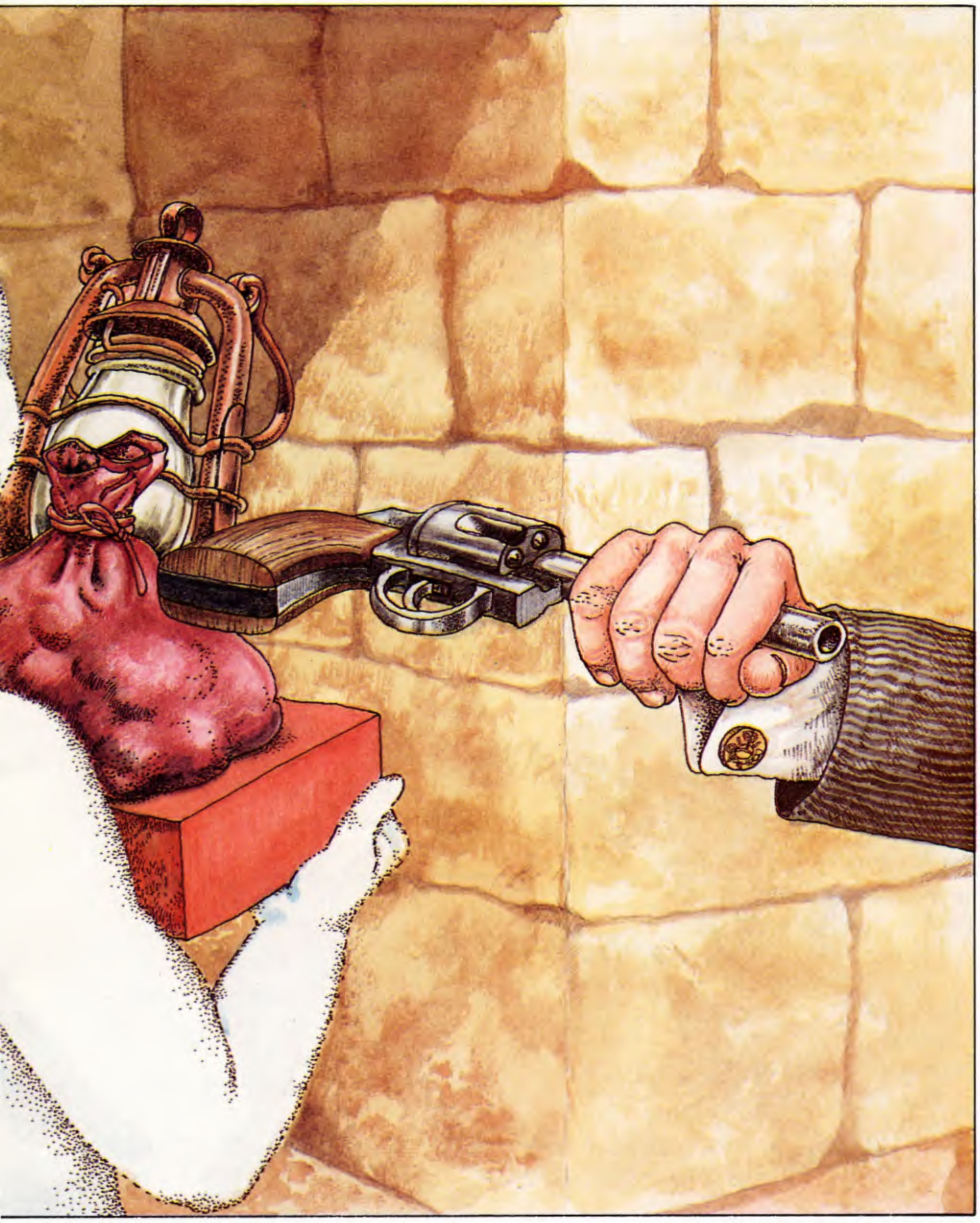


TAX BILLS: THE ANSWER

When the tax inspector rears his ugly head there's only one solution. The adventurer must shoot him with the gun which is to be found across the river:

```
1540 REM **SHOOT**
1550 IF B(4)<>-1 THEN PRINT "WITH
   WHAT?": GOTO 320
1560 IF B(7)<>L THEN PRINT
   V$;"□WHO?":GOTO 320
1570 PRINT "YOU KILLED THE□";B$(7): LET
   B(7)=0: GOTO 330
```

```
1540 REM **SHOOT**
1550 IF OB(4)<>-1 THEN PRINT
```



```
  "□WITH WHAT?":GOTO 32Ø
156Ø IF OB(7)< >L THEN PRINT
  V$;"□WHO":GOTO 32Ø
157Ø PRINT"□YOU KILLED THE□";
  OB$(7):OB(7)=Ø:
  GOTO 33Ø
```

The routine will be used when the adventurer uses the words KILL or SHOOT. If the gun isn't being carried (OB(4)< >−1 or B(4)< >−1), then Line 155Ø displays the message WITH WHAT? Similarly, if the tax inspector isn't present and the player tries to KILL, Line 156Ø asks WHO?

Line 157Ø tells the adventurer YOU KILLED THE TAX INSPECTOR, and adjusts the object location array so that the tax inspector no longer exists.

THE INSPECTOR'S REVENGE

The adventurer gets his come-uppance:

```
158Ø REM **TAX INSPECTOR**
159Ø LET IN=Ø: LET B(7)=Ø
16ØØ FOR K=1 TO NB
161Ø IF B(K)=−1 THEN LET IN=
  IN+1
162Ø NEXT K
163Ø IF IN=Ø THEN PRINT "AS YOU DIDN'T
  HAVE ANYTHING HE□□LOCKS YOU IN A
  DEEP DUNGEON": GOTO 136Ø
164Ø LET K=INT (RND*NB)+1: IF
  B(K)< >−1 THEN GOTO 164Ø
165Ø PRINT "HE TAKES THE□";B$(K),
  "AWAY FROM YOU": LET B(K)=Ø:
  GOTO 4ØØ
```

```
158Ø REM **TAX INSPECTOR**
159Ø IN=Ø:OB(7)=Ø
16ØØ FOR K=1 TO NB
161Ø IF OB(K)=−1 THEN IN=IN+1
162Ø NEXT
163Ø IF IN< >Ø THEN 164Ø
1635 PRINT "□AS YOU DIDN'T HAVE
  ANYTHING HE LOCKS"
1638 PRINT "□YOU IN A DEEP DUNGEON":
  GOTO 136Ø
164Ø K=INT(RND(1)*NB+1):IF
  OB(K)< >−1 THEN 164Ø
165Ø PRINT "HE TAKES THE□";
  OB$(K);"□AWAY FROM YOU":
  OB(K)=Ø:GOTO 33Ø
```

```
158Ø REM **TAX INSPECTOR**
159Ø IN=Ø:OB(7)=Ø
16ØØ FOR K=1 TO NB
161Ø IF OB(K)=−1 THEN IN=IN+1
162Ø NEXT
163Ø IF IN=Ø THEN PRINT"AS YOU DIDN'T
  HAVE ANYTHING HE LOCKED□□□YOU
  IN A DEEP DUNGEON":GOTO 136Ø
164Ø K=RND(NB):IF OB(K)< >−1 THEN
  164Ø
165Ø PRINT"HE TAKES THE□";
  OB$(K),"□AWAY FROM YOU":
  OB(K)=Ø:GOTO 33Ø
```

```
158Ø REM **TAX INSPECTOR**
159Ø IN=Ø:OB(7)=Ø
16ØØ FOR K=1 TO NB
161Ø IF OB(K)=−1 THEN IN=IN+1
162Ø NEXT
163Ø IF IN=Ø THEN PRINT"□AS YOU
  DIDN'T HAVE ANYTHING HE□□LOCKS
  YOU IN A DEEP DUNGEON":GOTO136Ø
164Ø K=RND(NB):IF OB(K)< >−1 THEN
  164Ø
165Ø PRINT"HE TAKES THE□";
  OB$(K),"□AWAY FROM YOU":
  OB(K)=Ø:GOTO 33Ø
```

The tax inspector is only allowed to appear once during the adventure, so Line 159Ø adjusts the object location array so that he doesn't exist as far as the program is concerned. This doesn't have any effect on this routine, but it saves him appearing at the next location as well. IN is a counter used for checking if objects are being carried.

Lines 16ØØ to 162Ø go through the object location array checking if each object in turn is being carried. Any object that is being carried increases IN by 1.

If nothing is being carried, then the value of IN remains at zero and the adventurer is told

AS YOU DIDN'T HAVE ANYTHING HE LOCKS YOU IN A DEEP DUNGEON. The game ends and the adventurer is asked if he wants another go by jumping to Line 136Ø.

If objects are being carried, then Line 164Ø picks one at random. As long as that number object is being carried it is confiscated. If it isn't being carried, then another object is chosen at random, until the selection tallies with one that is being carried.

If a suitable object has been picked Line 165Ø tells the adventurer that the object has been confiscated by the tax inspector. The object location array is altered so that the object no longer exists.

GOING FOR A DIP

This routine will be used when the adventurer decides to swim the river:

```
1400 REM **SWIM**
1410 IF L< >7 THEN PRINT "IN WHAT?!!":
    GOTO 400
1420 IF B(2)= -1 THEN PRINT "WHAT A
    SHAME, YOU DROWNED": GOTO 1360
1430 IF B(4)> -1 THEN PRINT "YOU FIND
    A GUN": LET B(4)= -1: GOTO 400
1440 PRINT "YOU GET WET":
    GOTO 400
```

```
1400 REM **SWIM**
1410 IF L< >7 THEN PRINT"□IN
    WHAT?!!":GOTO330
1420 IF OB(2)= -1 THEN PRINT"□WHAT A
    SHAME, YOU DROWNED":GOTO1360
1430 IF OB(4)> -1 THEN PRINT
    "□YOU FIND A GUN":OB(4)=
    -1:GOTO330
1440 PRINT"□YOU GET WET":
    GOTO330
```

Line 141Ø checks if the adventurer is by the river. If not, then the question IN WHAT?!! is posed. As there's no swimming pool or ocean in the adventure, there's no point in writing an input routine to handle any replies. No prompt appears, and the game carries on with the directions being displayed.

If the adventurer tries to swim the river while still clutching the brick, he dies—WHAT A SHAME, YOU DROWNED. After drowning he may be reincarnated when he is asked if wants another go.

Line 143Ø checks if the adventurer isn't carrying the gun. In that case the object location array is adjusted and the message YOU FIND A GUN appears.

If the adventurer has already found the gun and, for some reason, is trying to swim the river again, Line 144Ø says YOU GET WET.

AT LAST, THE EYEBALL

The adventurer can only recover the fabled eyeball if the bag of marbles has been found. The next step is to empty the bag and the eyeball will appear.

Here's the routine:

```
1450 REM **EMPTY**
1460 IF N$< >"BAG"( TO LEN N$) THEN
    PRINT "YOU CAN'T EMPTY THAT": GOTO
    400
1470 IF B(1)< > -1 THEN LET G=1: GOTO
    1270
1480 PRINT "THE MARBLES ROLL OVER THE
    FLOOR": LET B(5)=L: GOTO 370
```

```
1450 REM ** EMPTY **
1460 IN=0:IF N$=LEFT$("BAG",
    LEN(N$)) THEN IN=1
1465 IF IN< >1 THEN PRINT "□YOU CAN'T
    EMPTY THAT":GOTO 330
1470 IF OB(1)< > -1 THEN G=1:GOTO
    1270
1480 PRINT "□THE MARBLES ROLL OVER
    THE FLOOR":OB(5)=L: GOTO 370
```

```
1450 REM **EMPTY**
1460 IN=INSTR("BAG",N$):IF IN< >1
    THEN PRINT"□YOU CAN'T EMPTY
    THAT":GOTO330
1470 IF OB(1)< > -1 THEN G=1:GOTO
    1270
1480 PRINT"□THE MARBLES ROLL OVER
    THE FLOOR":OB(5)=L: GOTO 370
```

The routine is called when the adventurer instructs EMPTY *something*. Line 146Ø checks if that something is a bag. If it isn't (N$< >"BAG"), then the message YOU CAN'T EMPTY THAT appears. Line 147Ø checks whether the bag is present (OB(1)< > -1 or B(1)< > -1). If it isn't, instead of having another message, the program jumps to Line 127Ø to use the YOU HAVEN'T GOT IT message you have already included in the program.

If the bag is present the program reaches Line 148Ø. The message: THE MARBLES ROLL OVER THE FLOOR is PRINTed, and the object location array is adjusted because the eyeball is now at the current location.

There's no need to PRINT a message to this effect, because by jumping to Line 37Ø the usual long description mechanism can be used instead. The description that you have entered in the long description array—see Line 24Ø—appears on the screen.

LIGHTING THE LAMP

The lamp needs to be lit if the adventurer is to be able to see the exits from the darkened room. If the adventurer isn't carrying the lamp, the gloom won't be pierced and he'll find himself stuck. This is the lamp lighting routine:

```
1490 REM **LIGHT**
1500 IF N$< >"LAMP"( TO LEN N$) THEN
    PRINT "YOU CAN'T DO THAT": GOTO 400
1510 IF B(6)< > -1 THEN LET G=6: GOTO
    1270
1520 IF LA=1 THEN PRINT "IT'S ALREADY
    LIT": GOTO 400
1530 LET LA=1: LET DA=0: PRINT "OK":
    GOTO 330
```

```
1490 REM ** LIGHT **
1500 IN=0:IF N$=LEFT$("LAMP",
    LEN(N$)) THEN IN=1
1505 IF IN< >1 THEN PRINT "□YOU CAN'T
    DO THAT":GOTO 330
1510 IF OB(6)< > -1 THEN
    G=6:GOTO1270
1520 IF LA=1 THENPRINT "IT'S ALREADY
    LIT":GOTO 330
1530 LA=1:PRINT "OK":GOTO 330
```

```
1490 REM **LIGHT**
1500 IN=INSTR("LAMP",N$):IF IN< >1
    THENPRINT"□YOU CAN'T DO
    THAT":GOTO330
1510 IF OB(6)< > -1 THENG=6:
    GOTO1270
1520 IF LA=1 THEN PRINT"IT'S ALREADY
    LIT":GOTO330
1530 LA=1:PRINT"OK":GOTO 330
```

If the adventurer instructs LIGHT something the routine is called. Line 15ØØ (15Ø5 in the Commodore) is very similar to the equivalent line in the 'empty' routine, checking if the adventurer has named the lamp. The YOU CAN'T DO THAT message appears in exactly the same way as before.

Line 152Ø checks if the 'lamp alight' flag—LA—is set, and tells the adventurer if the lamp is already lit.

The lamp alight flag is set to 1 by Line 153Ø and it also PRINTs OK.

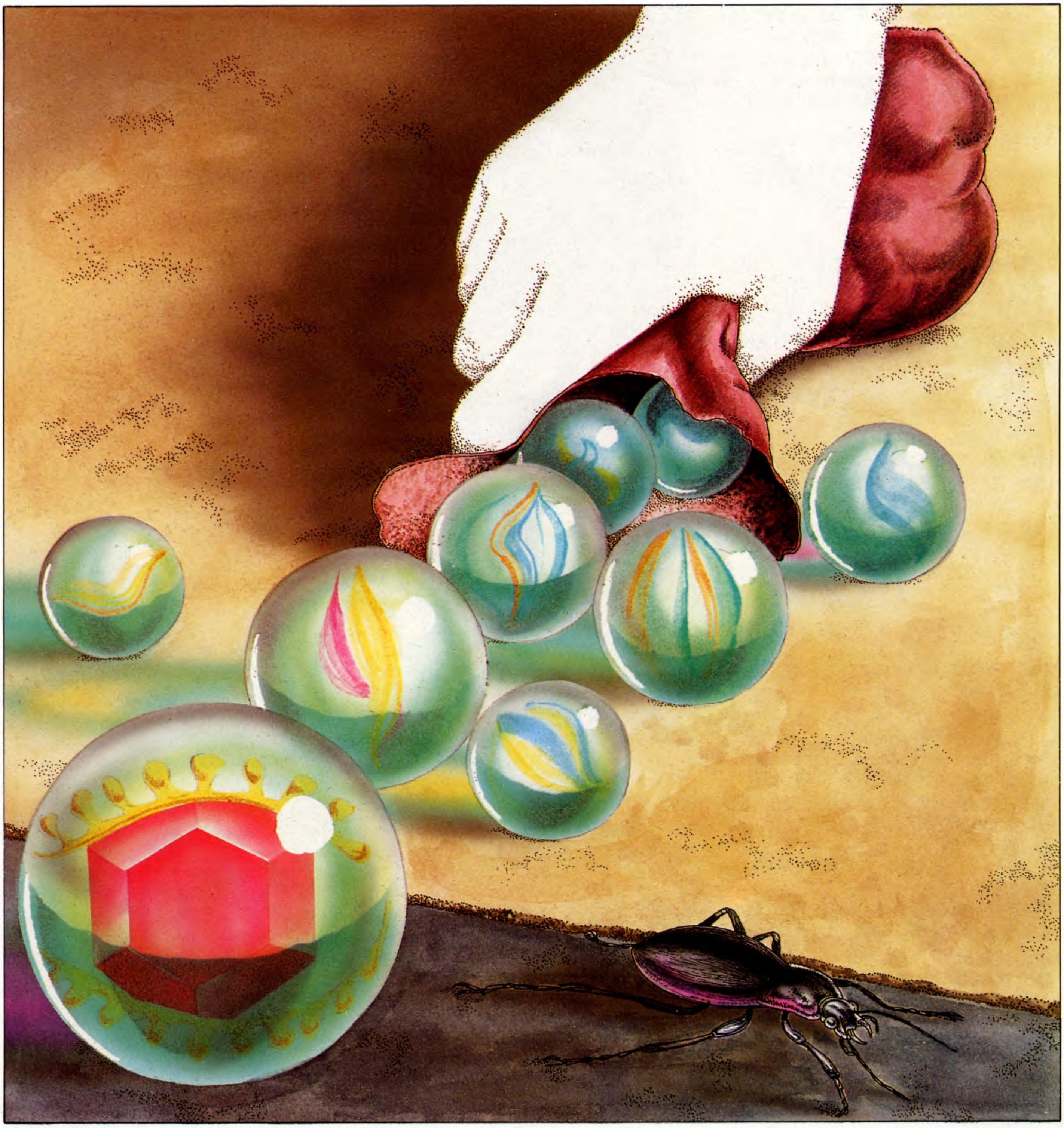
THE END IS NIGH

The chain is hanging in the throne room and the adventurer has arrived on the scene.

What should he do? How about pulling the chain? Here's a routine which will administer the consequences:

```
1300 REM **PULL**
1310 IF N$="CHAIN"( TO LEN N$) THEN
   LET IN=1: IF IN=1 AND L<>24 THEN
   PRINT "NOTHING HAPPENS": GOTO 400
1320 IF IN<>1 THEN PRINT "YOU CAN'T
   PULL THAT !": GOTO 400
1330 IF B(5)<>-1 THEN PRINT "YOU GET
   FLUSHED DOWN THE TOILET AND GO
   ROUND THE BEND": GOTO 1360
1335 REM **END OF ADVENTURE**
1340 PRINT "WELL DONE. YOU'VE
   COMPLETED THE ADVENTURE"
1360 PRINT '"DO YOU WANT ANOTHER
   GAME (Y/N)?"
1370 LET A$=INKEY$: IF A$<>"Y" AND
   A$<>"N" THEN GOTO 1370
1380 IF A$="Y" THEN RUN
1390 STOP
```

```
1300 ** PULL **
1310 IN=0:IF N$=LEFT$("CHAIN",
   LEN(N$))THEN IN=1
1315 IF IN=1 AND L<>24 THEN PRINT
   "NOTHING HAPPENS":GOTO 330
```



```
1320 IF IN< >1 THEN PRINT "□YOU CAN'T
    PULL THAT!":GOTO330
1330 IF OB(5)=-1 THEN 1340
1335 PRINT "□YOU GET FLUSHED DOWN
    THE TOILET AND GO ROUND THE
    BEND":GOTO 1360
1340 REM ** END OF ADVENTURE **
1350 PRINT "WELL DONE.YOU'VE
    COMPLETED THE ADVENTURE"
1360 PRINT:PRINT "▓DO YOU WANT
    ANOTHER GAME (Y/N)?"
1370 GET A$:IF A$< >"Y" AND
    A$< >"N" THEN 1370
1380 IF A$="Y" THEN RUN
1390 PRINT "◘◘":POKE 53280, 14:END
```

Vic 20 users should alter Line 139Ø to:

```
1390 PRINT "◘◘": POKE 36879,27:END
```

```
1300 REM **PULL**
1310 IN=INSTR("CHAIN",N$):IF
    IN=1 AND L< >24 THEN PRINT
    "□NOTHING HAPPENS":GOTO330
1320 IF IN< >1 THEN PRINT"□YOU
    CAN'T PULL THAT!":GOTO330
1330 IF OB(5)< >-1 THEN PRINT
    "□YOU GET FLUSHED DOWN THE TOILET
    AND GO ROUND THE BEND":
    GOTO1360
1340 REM **END OF ADVENTURE**
1350 PRINT"WELL DONE. YOU'VE
    COMPLETED THE ADVENTURE"
1360 PRINT:PRINT"□DO YOU WANT
    ANOTHER GAME (Y/N)?"
1370 A$=INKEY$:IF A$< >"Y" AND
    A$< >"N" THEN 1370
1380 IF A$="Y" THEN RUN
1390 END
```

Acorn users must use GET$ instead of INKEY$ in Line 137Ø.

Line 131Ø covers the possibility that the adventurer has taken the chain out of the throne room before pulling it. The line will tell the adventurer NOTHING HAPPENS.

If the adventurer tries to pull any other object in the adventure he's told YOU CAN'T PULL THAT! by Line 132Ø.

After that the unthinkable happens. If the adventurer is in the throne room, but hasn't found the eyeball—the message YOU GET FLUSHED DOWN THE TOILET AND GO ROUND THE BEND is displayed and the game ends.

If the adventurer has managed to find the eyeball and has pulled the chain in the throne room, none of these lines will have any effect and you can breathe a sigh of relief as it says WELL DONE. YOU'VE COMPLETED THE ADVENTURE.

Finally, there's an option for another game in Lines 136Ø to 138Ø. This is only really of use if the adventurer has got locked in the dungeon, or has been flushed down the toilet.

THE INSTRUCTIONS

You should now have a fully functioning adventure game, so it's time to add the finishing touches.

Without any instructions, the adventurer will not know the point of all your efforts, or what to do. Before you add the instructions routine to a game, check how much memory remains (see page 268). If there's only a small amount, now's the time to remove all the extraneous REM lines—although you will have to renumber the GOSUBs which direct the program to them, to avoid error reports.



The amount of instructions you give will have to be carefully considered. You will have to decide according to the amount of memory available, how many hints you wish to give at that stage, and perhaps even considerations like the screen format on your machine which will affect how much detail you can give before having to resort to another screen.

The *INPUT* adventure is very simple, so the instructions routine is short, and contains little information. Here it is:



```
80 CLS:PRINT "DO YOU WANT
   INSTRUCTIONS (Y/N)?"
90 LET A$=INKEY$:IF A$="" THEN GOTO
   90
95 IF A$="Y" THEN GOSUB 6000
6000 REM **INSTRUCTIONS**
6010 CLS:PRINT:PRINT"□□IN THE FACE
   OF FINANCIAL□□□□□□
   COLLAPSE YOU HAVE FLED THE
   □□□□□□ COUNTRY"
6020 PRINT:PRINT "□□THE SOLUTION TO
   YOUR PROBLEMS LIES IN FINDING THE
   FABLED□□□□□□
   JEWELLED EYEBALL OF THE PURPLE□□
   ICON AND PASSING THE FINAL
   □□□□□□INITIATIVE TEST"
6030 PRINT:PRINT "□□AVOID THE TAX
   INSPECTOR AT□□□□ALL COSTS"
6040 PRINT AT 20,3;"PRESS ANY KEY TO
   CONTINUE"
6050 LET A$=INKEY$:IF A$="" THEN
   GOTO 6050
6060 RETURN
```

```
10 PRINT"▢DO YOU WANT INSTRUCTIONS?"
20 GET A$:IF A$="" THEN 20
30 IF A$="Y" THEN GOSUB 6000
6000 REM**INSTRUCTIONS**
6010 PRINT:PRINT "▢□□IN THE FACE OF
   FINANCIAL COLLAPSE YOU HAVE FLED
   THE COUNTRY"
6020 PRINT:PRINT"□□THE SOLUTION TO
   YOUR PROBLEMS LIES IN"
6025 PRINT "FINDING THE FABLED
   JEWELLED EYEBALL OF"
6027 PRINT"THE PURPLE ICON AND
   PASSING THE FINAL"
6029 PRINT "INITIATIVE TEST"
6030 PRINT:PRINT"□□AVOID THE TAX
   INSPECTOR AT ALL COSTS"
6040 PRINT TAB(8) "▪PRESS ANY KEY TO
   CONTINUE"
6050 GET A$:IF A$="" THEN 6050
6060 RETURN
```

```
1 CLS:PRINT "DO YOU WANT
   INSTRUCTIONS (Y/N)?"
20 A$=GET$
30 IF A$="Y" THEN GOSUB 6000
6000 REM**INSTRUCTIONS**
6010 CLS:PRINT' "□□IN THE FACE OF
   FINANCIAL COLLAPSE YOU HAVE FLED
   THE COUNTRY"
6020 PRINT' "□□THE SOLUTION TO YOUR
   PROBLEMS LIES IN FINDING THE FABLED
   JEWELLED EYEBALL OF□□THE PURPLE
   ICON AND PASSING THE
   FINAL□□□INITIATIVE TEST"
6030 PRINT' "AVOID THE TAX INSPECTOR AT
   ALL COSTS"
6040 PRINT TAB(8,23) "PRESS ANY KEY TO
   CONTINUE"
6050 A$=GET$
6060 RETURN
```

```
10 CLS:PRINT "DO YOU WANT
   INSTRUCTIONS (Y/N)?"
20 A$=INKEY$:IF A$="" THEN 20
30 IF A$="Y" THEN GOSUB 6000
6000 REM**INSTRUCTIONS**
6010 CLS:PRINT:PRINT"□□IN THE
   FACE OF FINANCIAL□□□□□□
   COLLAPSE YOU HAVE FLED
   THE□□□□□□COUNTRY"
6020 PRINT:PRINT "□□THE SOLUTION
   TO YOUR PROBLEMS LIES IN FINDING
   THE FABLED□□□□□□JEWELLED
   EYEBALL OF THE PURPLE□□ICON
   AND PASSING THE FINAL□□□□□□
   INITIATIVE TEST"
6030 PRINT:PRINT "□□AVOID THE TAX
   INSPECTOR AT□□□□ALL COSTS"
6040 PRINT@451, "PRESS ANY KEY TO
   CONTINUE"
6050 A$=INKEY$:IF A$="" THEN 6050
6060 RETURN
```

Now SAVE the completed adventure on tape.

Next time you'll see how to use the structure that you have followed through the Jewelled Eyeball of the Purple Icon as a basis for your own adventures.

Microtip

Brighten up the adventure

Once you've completed writing the adventure game you can think about adding a title page at the start, and a final congratulations page.

How much you can do depends on the amount of memory left—see page 268—but you should at least be able to add a little colour.

The title page should be eye catching enough to make you want to play the game while the congratulations page should make you feel it was worthwhile getting that far—perhaps some simple graphics or even a short tune.

GETTING THINGS IN ORDER

Every type of data handling program can benefit from using a sort routine of one type or another. Here we look at three of the more common methods of sorting



Sorting is one of the fundamental aspects of information processing. Computers are very good at manipulating data quickly, and sorting plays a very important part in making this information accessible for inspection and amendment.

Imagine how difficult it would be to trace every mention of a particular subject in a text book without the aid of an index. Or to look up a telephone number if the directory wasn't arranged in alphabetical order. Both are essentially lists of information but—unlike the haphazard entries of something like a shopping list—items are arranged, or sorted, in a specific order: alphabetically. But you could just as well look at items which are sorted numerically—cheque entries in a bank statement is an example.

Many types of program make use of sorting routines—even games which feature a 'Hall of Fame' high score display.

But sorting routines are more usually found on programs which have to handle information lists—such as datafile records, mail label printouts, and the like.

WHAT IS SORTING?

Sorting is simply the action of placing information in a specified order—like fanning out and ordering a handful of playing cards.

The relative placing of any two items is usually based on the numerical or alphabetical superiority of one of them.

In numerical sorts high values come successively either before or after lower values. In alphabetical sorts, letters nearer the beginning of the alphabet are placed either before or after those which come nearer the end. It's also possible to base your sort on an alphanumeric order, where letters and numbers both count, but the priority of one must always come before the other—letters usually having greater importance than numbers.

In numerical sorts, the value 1 usually has the highest priority, with A assuming the prime spot in alphabetical sorts. But for special purposes these priorities can be reversed—for example, to give the value 9 and the letter Z the prime spots.

THE BUBBLE SORT

The simplest of the recognised sort routines is described as the bubble sort. You have seen one example of this already, on pages 216 to

■ WHAT IS SORTING?
■ THE BUBBLE SORT—SIMPLE BUT PAINFULLY SLOW
■ THE BUBBLE SORT ROUTINE EXPLAINED
■ THE ADVANTAGES OF THE BINARY SORT ROUTINE
■ SHELL AND SHELL-METZNER PROGRAM MODULES
■ IMPROVING SORT SPEEDS



219. Now, let's look at the process in more detail. As an exercise, use some playing cards to follow what happens. Or cut out and number some pieces of paper.

Place this selection of cards on a table in this order, with the 'three' card furthest away from you:

last [THREE]
[SEVEN]
[TWO]
[SIX]
first [TEN]

In a bubble sort, the first value—here a 10 card—is compared with the next one along in the array, interchanging if the next one is (in this case) lower in value. The 10 is then in turn compared with the 2, the 7 and the 3. Each time, the 10 changes places—because it is the higher of the two values.

In effect, the 10—always the higher value of any compared pair—has 'bubbled' through the rest of the values, which explains the most popular name for this sort routine. By the same token, low values also bubble downwards. The bubble sort is also known as an exchange or interchange sort.

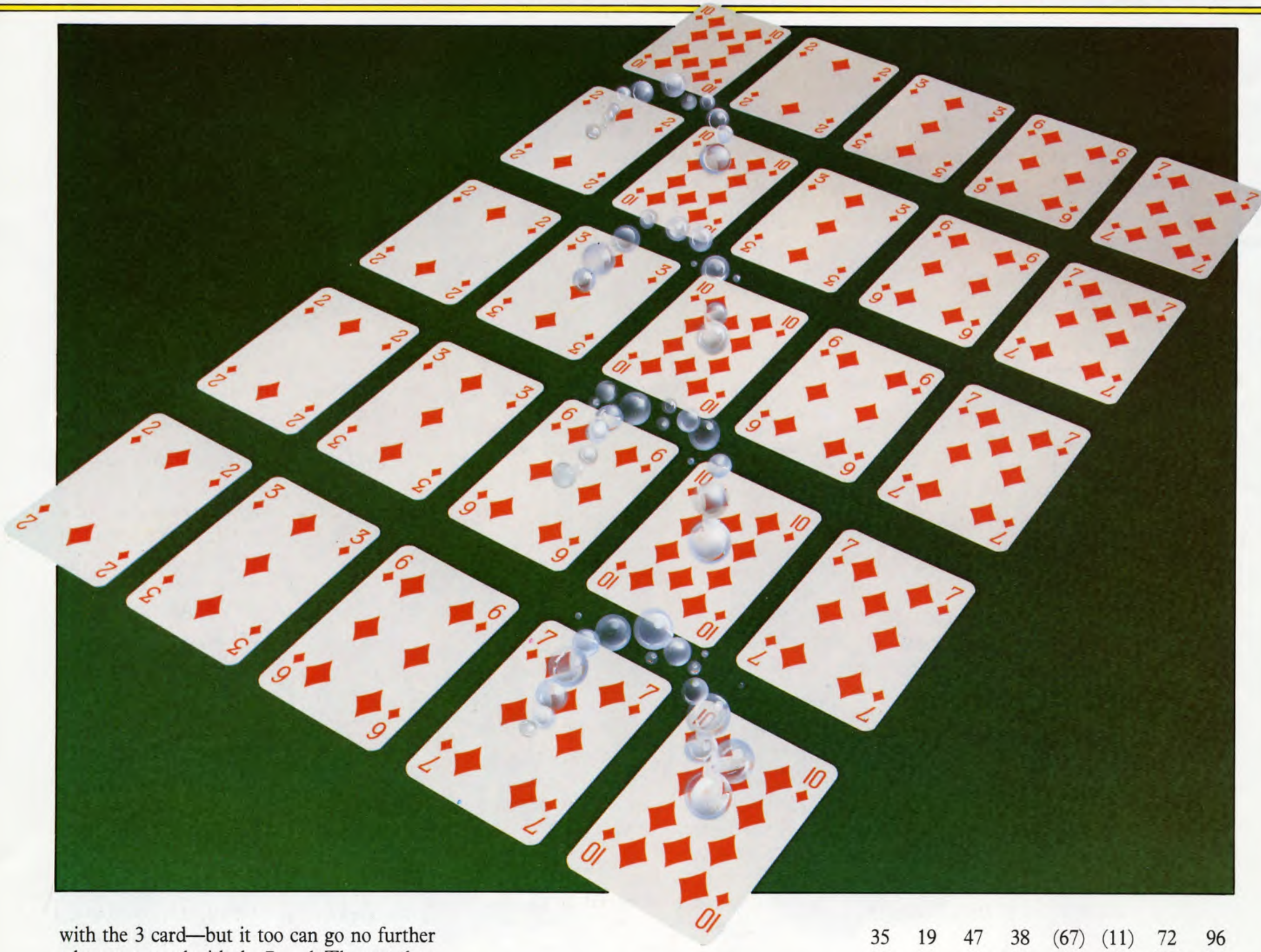
So after one pass the card arrangement looks like:

last [TEN]
[THREE]
[SEVEN]
[TWO]
first [SIX]

The process then starts again with the new 'first' card, 6, which is compared with 2 and exchanged because it is higher in value. But then it is compared with 7. Now 7 is obviously higher and the sort now continues with this value, leaving 6 in its last successful position. The 7 card is compared first with 3 and then with 10, moving up only one place because it too has met a higher value. The card order now looks like:

last [TEN]
[SEVEN]
[THREE]
[SIX]
first [TWO]

The new first card, 2, is then compared with its neighbour, 6. The 6 card then takes over because it is the higher value, interchanging



with the 3 card—but it too can go no further when compared with the 7 card. The sort then comes to an end, as no further exchanges can take place, and the card order looks like:

last	[TEN]
	[SEVEN]
	[SIX]
	[THREE]
first	[TWO]

A computer would in fact conduct a final run through this new card order to see whether or not any card pair could be interchanged—and only stop when they could not.

During the course of the bubble sort a number of comparisons and exchanges take place. These two distinct operations are part of any sort routine and the difference between routines rests entirely on the numbers of each and the time taken.

Let's now look at an example using this sequence of numbers:

top:							**bottom:**
67	35	72	19	47	38	11	96

Top (first) in this list is 67. This is compared and exchanged with 35 but gets no further. The higher value at this point, 72, is compared and exchanged in turn with 19, 47, 38, 11 until it meets 96 when the routine starts again at the beginning with this first number. In this list below, which shows the procedure step by step, items subjected to a comparison are enclosed. If a different number appears in the row below, an exchange has occurred.

top:							**bottom:**
(67)	(35)	72	19	47	38	11	96
35	(67)	(72)	19	47	38	11	96
35	67	(72)	(19)	47	38	11	96
35	67	19	(72)	(47)	38	11	96
35	67	19	47	(72)	(38)	11	96
35	67	19	47	38	(72)	(11)	96
35	67	19	47	38	11	(72)	(96)
(35)	(67)	19	47	38	11	72	96
35	(67)	(19)	47	38	11	72	96
35	19	(67)	(47)	38	11	72	96
35	19	47	(67)	(38)	11	72	96
35	19	47	38	(67)	(11)	72	96
35	19	47	38	11	(67)	(72)	96
35	19	47	38	11	67	(72)	(96)
(35)	(19)	47	38	11	67	72	96
19	(35)	(47)	38	11	67	72	96
19	35	(47)	(38)	11	67	72	96
19	35	38	(47)	(11)	67	72	96
19	35	38	11	(47)	(67)	72	96
19	35	38	11	47	(67)	(72)	96
19	35	38	11	47	67	(72)	(96)
(19)	(35)	38	11	47	67	72	96
19	(35)	(38)	11	47	67	72	96
19	35	(38)	(11)	47	67	72	96
19	35	11	(38)	(47)	67	72	96
19	35	11	38	(47)	(67)	72	96
19	35	11	38	47	(67)	(72)	96
19	35	11	38	47	67	(72)	(96)
(19)	(35)	11	38	47	67	72	96
19	(35)	(11)	38	47	67	72	96
19	11	(35)	(38)	47	67	72	96
19	11	35	(38)	(47)	67	72	96
19	11	35	38	(47)	(67)	72	96
19	11	35	38	47	(67)	(72)	96
19	11	35	38	47	67	(72)	(96)

(19)	(11)	35	38	47	67	72	96
11	19	35	38	47	67	72	96

So how does a bubble sort look in program form? Try this program, in which the bubble sort routine occupies the subroutine from Line 1ØØØ onwards. SAVE the program as other sort routines can be added to it:

```
10 POKE 23658,8: LET T=Ø: INPUT
   "NUMBER OF ITEMS□";AA: IF AA<2
   THEN GOTO 1Ø
15 DIM A(AA)
2Ø PRINT "UNSORTED TABLE": PRINT
3Ø FOR Z=1 TO AA
4Ø LET A(Z)=INT (RND*1ØØ)+1
5Ø PRINT TAB T; A(Z);: LET T=T+4: IF
   T>3Ø THEN LET T=Ø
6Ø NEXT Z
7Ø PRINT : PRINT : PRINT "PRESS S FOR
   SORT"
8Ø LET K$=INKEY$: IF K$<>"S" THEN
   GOTO 8Ø
9Ø GOSUB 1ØØØ
1ØØ PRINT : PRINT "SORTED TABLE": PRINT
11Ø LET T=Ø: FOR Z=1 TO AA
12Ø PRINT TAB T;A(Z);: LET T=T+4: IF
   T>3Ø THEN LET T=Ø
13Ø NEXT Z
14Ø GOTO 1Ø
999 REM BUBBLE SORT
1ØØØ FOR Z=1 TO AA-1
1Ø1Ø LET ZZ=Ø
1Ø2Ø FOR Y=1 TO AA-Z
1Ø3Ø IF A(Y+1)<A(Y) THEN LET X=A(Y):
   LET A(Y)=A(Y+1): LET A(Y+1)=X: LET
   ZZ=1
1Ø4Ø NEXT Y
1Ø5Ø IF ZZ=Ø THEN RETURN
1Ø6Ø NEXT Z: RETURN
```

```
1Ø PRINT: PRINT: INPUT "NUMBER OF
   ITEMS"; AA: IF AA<2 THEN 1Ø
15 DIM A(AA)
2Ø PRINT: PRINT "UNSORTED TABLE": PRINT
3Ø FOR Z=1 TO AA
4Ø A(Z)=INT(RND(1)*1ØØ)+1
5Ø PRINT A(Z),
6Ø NEXT Z
7Ø PRINT: PRINT: PRINT "PRESS S FOR
   SORT"
8Ø GET K$: IF K$<>"S" THEN 8Ø
9Ø GOSUB 1ØØØ
1ØØ PRINT: PRINT: PRINT "SORTED  TABLE"
11Ø PRINT: FOR Z=1 TO AA
12Ø PRINT A(Z),
13Ø NEXT Z
14Ø RUN
999 REM *** BUBBLE SORT ***
1ØØØ FOR Z=1 TO AA-1
1Ø1Ø ZZ=Ø
1Ø2Ø FOR Y=1 TO AA-Z
1Ø3Ø IF A(Y+1)<A(Y) THEN X=A(Y):
   A(Y)=A(Y+1): A(Y+1)=X: ZZ=1
1Ø4Ø NEXT Y
1Ø5Ø IF ZZ=Ø THEN RETURN
1Ø6Ø NEXT Z: RETURN
```

Make the following amendments for your machine:

To make sure the INPUT routines work properly on your model, use commas rather than semicolon after the prompt in Line 1Ø and change Lines 4Ø and 8Ø:

```
1Ø INPUT '' "NUMBER OF ITEMS",AA: IF
   AA<2 THEN 1Ø
4Ø A(Z)=RND(1ØØ)
8Ø K$=GET$: IF K$<>"S"THEN8Ø
```

```
4Ø A(Z)=RND(1ØØ)
5Ø PRINT A(Z);
8Ø K$=INKEY$: IFK$<>"S"THEN 8Ø
12Ø PRINT A(Z);
```

RUN the program. It starts by asking you how many items you wish to sort, creating a set of pseudo-random numbers based on your reply. These are displayed on the screen under the heading 'Unsorted Table'. A prompt then asks you if you are ready to start the sort—simply press 'S' to start it off.

Sorting then takes place. With a few items this may take only a second or so—but try entering 50 in reply to the opening prompt. When the sort is complete, the new group of figures are displayed after the others.

Further details of how the bubble sort works are given on pages 216 to 219. Look also at the algorithm shown in diagrammatic form on page 219.

For comparison purposes, it's helpful to time the sorting period. Timing is a job the computer can do for you, so incorporate this timing routine within the existing program:

```
9Ø POKE 23672,Ø: POKE 23673,Ø:
   GOSUB 1ØØØ: PRINT: PRINT(PEEK
   23672+256*PEEK 23673)/5Ø;"□
   SECONDS"
```

```
9Ø TI$="ØØØØØØ":GOSUB 1ØØØ:PRINT:
   PRINT "TIME:"; TI/6Ø; "SECONDS"
```

```
9Ø TIME=Ø: GOSUB 1ØØØ: PRINT
   '"TIME:□";TIME/1ØØ; "□SECONDS"
```

```
9Ø TIMER=Ø: GOSUB 1ØØØ: PRINT:
   PRINT"□TIME:□"; TIMER/5Ø;
   "SECONDS"
```

To start the timing process—and the sort routine—press S when the prompt appears. The elapsed time when the sort is completed is displayed and the program automatically reRUNs.

While the sort times are probably much faster than you could manage sorting the numbers yourself, by computer standards the bubble sort routine is painfully slow. And that is why it is not often used in its 'raw' state for data handling programs except on very short lists.

But, surprisingly, there is one instance when the bubble sort actually proves the fastest of the lot, and that's when it is used to re-sort a sorted list to which an item has been added.

In a mail-label program for instance, the new entries can be sorted separately and then

Microtip

The speed of any sort routine can be given a considerable boost if information is presented to it in a partly sorted form—this is especially true of the simple exchange or bubble sort.

If items have to be sorted in date order, for instance, it helps to input data in the form Year/Month/Day rather than in the more usual Day/Month/Year form. The routine could then very quickly work through Years to produce a coarse listing. This list can then be ordered again according to Month within each of those Years. Finally—in the area of greatest permutation—Days could be placed in order within the months.

Using this method, sorting is carried out first on the whole group, then on individual parts. At any one stage, all entries are in some sort of order.

This is rather faster than asking a routine to look through—and order—all the Days, then ask it re-order some of these because of the influence of the Months, and then once again because of the influence of Years.

An additional subroutine could be used to convert a 'normal' input sequence into a 'reverse order' one for sorting purposes.

merged with the old list. Alternatively, single new entries could be inserted in the old list, and the whole lot subjected to another sort.

Under these circumstances, the bubble sort has little more to do than compare the one new entry with all its neighbours until its correct position is found. And this can be very quick—something used to good advantage in other types of sort routine.

SHELL SORT

The so-called Shell sort is one of two popular alternatives to the bubble sort, either of which should be considered if the number of items to be sorted exceeds about ten. (Note that comparing a single new entry with a ready-sorted list means just two items are being compared.)

The Shell sort uses a binary search technique which works by successively splitting the original, unsorted, list into halves until the relative position of the new entry can be established. On each occasion, a new set of value pairs is presented for comparison until a complete pass is made with no swaps.

A Shell sort module typically takes the form shown here. Add this subroutine to your existing program and change the GOSUB reference in the middle of Line 9Ø to try it out:

```
1999 REM SHELL SORT
2ØØØ LET Z=AA
2Ø1Ø IF Z<=1 THEN RETURN
2Ø2Ø LET Z=INT (Z/2): LET Y=AA-Z
2Ø3Ø LET ZZ=Ø
2Ø4Ø FOR X=1 TO Y
2Ø5Ø LET XX=X+Z
2Ø6Ø IF A(X)>A(XX) THEN LET YY=A(X):
   LET A(X)=A(XX): LET A(XX)=YY: LET
   ZZ=1
2Ø7Ø NEXT X
2Ø8Ø IF ZZ>Ø THEN GOTO 2Ø3Ø
2Ø9Ø GOTO 2Ø1Ø
```

```
1999 REM *** SHELL SORT ***
2ØØØ Z=AA
2Ø1Ø IF Z<=1 THEN RETURN
2Ø2Ø Z=INT(Z/2): Y=AA-Z
2Ø3Ø ZZ=Ø
2Ø4Ø FOR X=1 TO Y
2Ø5Ø XX=X+Z
2Ø6Ø IF A(X)>A(XX) THEN YY=A(X):
   A(X)=A(XX): A(XX)=YY: ZZ=1
2Ø7Ø NEXT X
2Ø8Ø IF ZZ>Ø THEN 2Ø3Ø
2Ø9Ø GOTO 2Ø1Ø
```

So how's it work? The easiest way to see what is going on is by working through an example in detail. Suppose a list of numbers is presented in this order for initial sorting:

```
top:                         bottom:
67   35   72   19   47   38   11   96
```

In a Shell sort, the list is first split into two, and the values of the first item in each of the two groups are compared. An exchange takes place if the value nearer the start is higher. Splitting the group gives start values of 67 and 47, which are compared and then exchanged:

```
top:                           bottom:
47   35   72   19 : 67   38   11   96
```

After an exchange the next pair of values—here 35 and 38—are compared. No exchange takes place. Then 72 and 11 are compared, and exchanged. Finally 19 and 96 are compared. The list now looks like this:

```
top:                           bottom:
47   35   11   19 : 67   38   72   96
```

Each half is at this point divided again:

```
top:                               bottom:
47   35 : 11   19 : 67   38 : 72   96
```

The first value, 47, is compared with the third place value, 11, and exchanged. Second place value 35 is compared with fourth place 19 and they are exchanged. 35 now assumes fourth spot. The value in the third slot, 47, which has been subjected to a successful comparison, is compared with 67—but this time unsuccessfully. The fourth value, now 35, is compared with 38 but remains in place. The fifth value—67—is compared with seventh place—72—but remains in place. In sixth spot, 38 is compared with 96 with no change. The sequence is shown below. As before, values subjected to a comparison are enclosed: look to the line immediately below to check whether or not an exchange has taken place.

```
top:                                   bottom:
[47]   35 :[11]   19 : 67    38 : 72    96
          :           :          :
 11  [35]: 47   [19]: 67    38 : 72    96
          :           :          :
 11   19 :[47]   35 :[67]   38 : 72    96
          :           :          :
 11   19 : 47   [35]: 67   [38]: 72    96
          :           :          :
 11   19 : 47    35 :[67]   38 :[72]   96
          :           :          :
 11   19 : 47    35 : 67   [38]: 72   [96]
          :           :          :
 11   19 : 47    35 : 67    38 : 72    96
```

The list is now further divided to yield the following groups:

```
top:                                   bottom:
11 : 19 : 47 : 35 : 67 : 38 : 72 : 96
```

The values are again compared, with lower values nearer the start giving way only to higher values nearer the end. Note that the sort compares adjacent blocks only. First is compared with second, second with third, and so on throughout the list:



```
top:                                      bottom:
[11]:[19]: 47 : 35 : 67 : 38 : 72 : 96
    :    :    :    :    :    :    :
 11 :[19]:[47]: 35 : 67 : 38 : 72 : 96
    :    :    :    :    :    :    :
 11 : 19 :[47]:[35]: 67 : 38 : 72 : 96
    :    :    :    :    :    :    :
 11 : 19 : 35 :[47]:[67]: 38 : 72 : 96
    :    :    :    :    :    :    :
 11 : 19 : 35 : 47 :[67]:[38]: 72 : 96
    :    :    :    :    :    :    :
 11 : 19 : 35 : 47 : 38 :[67]:[72]: 96
```

The sort continues through the end values which are already in order, and then back to the beginning:




```
[11]:[19]: 35 : 47 : 38 : 67 : 72 : 96
  :    :    :    :    :    :    :
 11 :[19]:[35]: 47 : 38 : 67 : 72 : 96
  :    :    :    :    :    :    :
 11 : 19 :[35]:[47]: 38 : 67 : 72 : 96
  :    :    :    :    :    :    :
 11 : 19 : 35 :[47]:[38]: 67 : 72 : 96
  :    :    :    :    :    :    :
 11 : 19 : 35 : 38 :[47]:[67]: 72 : 96
```

Although the list is now sorted, the program would zip through once again. Any new value can then be compared with this ordered list in the same way as an ordinary bubble search—for that is in essence what occurs at every pair comparison.

SHELL-METZNER SORT

The Shell-Metzner sort is an even speedier binary sort routine which is derived from the Shell sort algorithm:

```
2999 REM SHELL-METZNER SORT
3000 LET Z=AA
3010 LET Z=INT (Z/2)
3020 IF Z=0 THEN RETURN
3030 LET Y=AA-Z: LET ZZ=1
3040 LET X=ZZ
3050 LET XX=X+Z
3060 IF A(X)<=A(XX) THEN GOTO 3090
3070 LET W=A(X): LET A(X)=A(XX): LET
   A(XX)=W: LET X=X-Z
3080 IF X>=1 THEN GOTO 3050
3090 LET ZZ=ZZ+1
3100 IF ZZ<=Y THEN GOTO 3040
3110 GOTO 3010
```

```
2999 REM *** SHELL-METZNER SORT ***
3000 Z=AA
3010 Z=INT(Z/2)
3020 IF Z=0 THEN RETURN
3030 Y=AA-Z: ZZ=1
3040 X=ZZ
3050 XX=X+Z
3060 IF A(X)<=A(XX) THEN 3090
3070 W=A(X): A(X)=A(XX): A(XX)=W:
   X=X-Z
3080 IF X>=1 THEN 3050
3090 ZZ=ZZ+1
3100 IF ZZ<=Y THEN 3040
3110 GOTO 3010
```

Add this routine to your existing sort demonstration program, adjusting the GOSUB reference in Line 90 to try it out.

As you can see, it is extremely fast by comparison with the bubble sort when you are dealing with even fifty random numbers. But as an experiment, try some really large sets of numbers. Make yourself a cup of coffee or something while the bubble sort is at work!

THE SPEAKING COMPUTER

There are now speech synthesisers for most home computers, so here's a guide to the two main types available with some hints on how to make them sound more realistic

In science fiction stories, computers have always been able to speak. There was Hal, the megalomaniac computer in *2001*, Marvin the Paranoid Android in *Hitch-hiker's Guide to the Galaxy*, and many many more. But it's only recently that real computers, and more specifically the home micro, have been able to have their say. The expansion of speech in the home computer market is thanks to the number of cheap synthesisers that have been introduced over the past year or so.

Speech synthesisers are available for all the computers covered in *INPUT*, although Spectrum, Commodore 64 and Vic owners have more choice than the others.

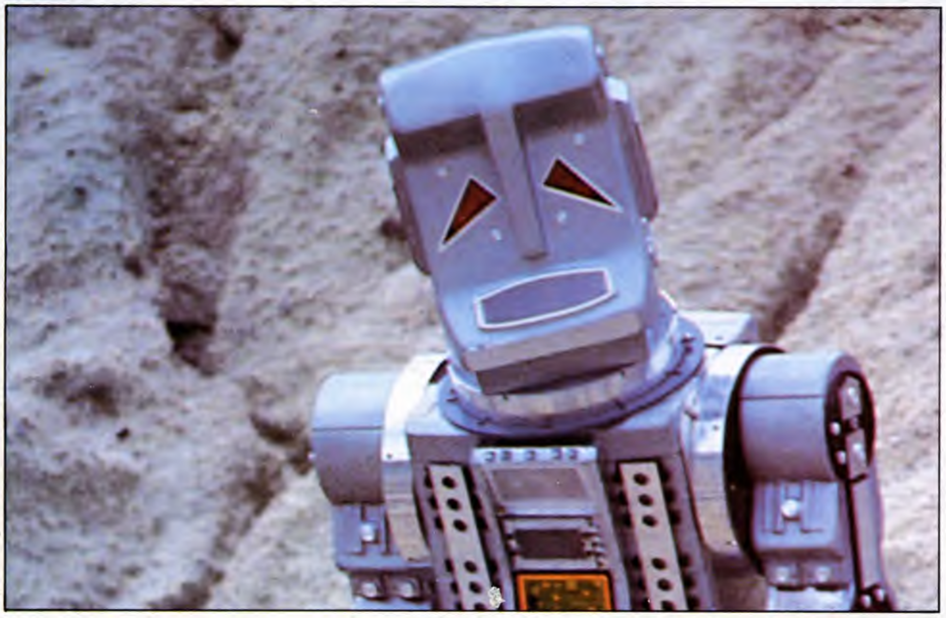
WHAT USE ARE THEY?

But why should you want your micro to speak? Well, the main reason is because it's such fun. Think of all the things you could get the computer to say—it could tell jokes, recite a poem, or even whisper sweet nothings into someone's ear. Or perhaps (and more practically) it could read out instructions for some intricate job you were doing to save you referring to a book all the time. The possibilities are endless.

An obvious use for speech synthesisers is in games. Until now, any message or instructions to the player had to be printed on the screen, which meant leaving sufficient space on the screen and so reducing the area for the game, or stopping the game completely while the message was printed. Now the computer can simply tell you what keys to press or how well you're doing.

Synthesisers can also make the game more fun by commenting on your play, such as 'well done' or 'watch out for the mutants' or 'you made a wrong move there'. In fact there are already quite a few games that make use of speech synthesisers and there are sure to be more to come. The synthesisers use up very little of your computer's memory so all your old games should still be able to RUN with the synthesiser fitted.

Speech synthesisers have serious uses too, of course, and one of the most exciting is that they make it much easier for a blind person to use a computer. The synthesiser can be made to speak each character or word as it is typed in, and it can also read out listings as well as prompts or instructions that appear on the screen.



Speech chips are now also used in cars—to remind you to put on your seat belt or to warn you when the oil pressure is low, and so on. And no doubt they'll soon be in all sorts of domestic appliances and industrial machinery as well—there are already some answering systems for taking telephone messages which use them.

They are also helpful in education, making it fun for a child to learn how to spell or recognize letters and numbers. There are already several spelling toys that speak a word and ask the child to type it in correctly. The computer then says whether the answer was right or wrong.

You can now do exactly the same job on your home micro. The advantage of using a speech synthesiser with small children is that the programs don't have to rely on the child being able to read any instructions.

TWO TYPES

There are two quite different approaches to speech synthesis. Both of these have advantages and disadvantages. One method uses actual samples of human speech—whole words, spoken letters and so on which are digitized and then stored on a chip, while the other type stores only single sounds produced by a sound generator in the computer, which then have to be strung together to make up words. Stored sample types have better sound quality, but are less efficient in memory use.

The most famous of the stored sample type is the Acorn speech synthesiser which uses sounds digitized from the voice of Kenneth Kendall, the newsreader. The BBC micro speaks with a true BBC accent!

The voice signal is first converted from an analogue to a digital form so it can be stored on the chip. Then, when the word or phrase is required, the digital signal is converted back into its original analogue form. In effect, it is

Writers who described fictional speaking computers would probably hardly believe the truth of their predictions—with computer 'voices' the size of a matchbox

■ WHY USE A SYNTHESISER?
■ STORED SPEECH TYPE
■ ALLOPHONE TYPE
■ WHAT ARE THE ALLOPHONES?
■ ADVANTAGES OF EACH TYPE
■ CONNECTING UP THE SYNTHESISER
■ PROGRAMMING IT TO SPEAK
■ IMPROVING PRONUNCIATION
■ LIST OF ALLOPHONES

not the voice signal that's stored, just information about that signal. Also, the information is stored in a very compact form. If the sound were simply digitized and stored normally, you'd need over 500K to store the two minutes of speech on the Acorn chip—rather more than the average home computer has! Luckily this can be compressed by about 98% so all the information can be contained on a normal ROM chip.

The second type of speech synthesiser is much more common. Instead of storing whole words it just stores single sounds like 't' or 'sh' or 'oo'. There are about 60 or so of the different sounds (called allophones) and they



are all that are needed to make up any word in the English language.

In this case it is not actual speech that is stored, just digital information that is used to control the sound generator to produce the relevant sound. Each particular sound has its own address which is stored in a look-up table in the chip. Any programs written for the synthesiser simply call up the sound whenever it is required. So instead of storing all the information needed to reconstruct a whole word, the memory only need store references to individual sounds. For one second of speech, only about 12 bytes need be reserved so even a whole sentence is unlikely to take up much of the computer's memory.

You may be familiar with the phonetic spelling of words in dictionaries that indicate how words should be pronounced. Sometimes phrase books for a foreign language do the same thing. If you've got one of these books have a look at the front as there's usually a list of all the different sounds used along with their spellings. The spelling used for each sound is often the same as the letters—t, b and so on—but there are a lot of complex sounds too like ng and ks. If you count them all up you'll find that there are about 50 or 60, although the exact number depends on the language you speak, and sometimes even on a regional accent. These are the same as the allophones used in the speech synthesisers. Only the spelling of the sounds may vary from one synthesiser to the next.

VOCABULARY?

The allophone type has the advantage of an unlimited vocabulary—any English word can be constructed from the individual allophones. The disadvantage is that the words do tend to sound rather robot-like and some words are very difficult to understand at first—although you soon learn to recognize the words after a bit of practice.

The stored sample type can only store about 100 words, plus the letters of the alphabet, numbers and a few suffixes and prefixes like -ing, in- and -teen. The stored sample synthesiser can't be reprogrammed with a different vocabulary, so it is a lot less flexible than the other type, although you may be able to buy extra vocabulary chips. However the sound of the words is more realistic and so is probably better where only short messages are needed and the clarity of the message is important—for example, in instructions like 'press any key to start' or warnings such as 'number too big'. It may also be more acceptable to people unused to using a computer who may be put off by a strange robot-sounding machine voice.

CONNECTING IT UP

Most speech synthesisers come enclosed in a black box and fit on the edge connector at the back of the computer or plug into one of the sockets. Several of them use a standard RS232 serial interface and so can be used with a number of different computers. Some have extension sockets themselves, so joysticks or other peripherals can be plugged in as well and used at the same time. A few exceptions do not fit directly onto the computer. One for the Vic 20 comes as a kit which you have to make up yourself, and then the completed board slots into the cartridge port. The Acorn synthesiser is just a set of chips which a dealer has to fit inside the computer itself. There is also a synthesiser for the Commodores that comes on a disk, needing no hardware at all.

Apart from the last three, connecting up most synthesisers is really just a matter of plugging them in.

Some of the plug-in ones have leads that you can connect to your TV set so the sound is routed through the TV speaker. Others can be connected to a hi-fi system. Both these methods have the advantage that you can control the volume of the sound, and on the hi-fi system you can control the tone too. It all helps to make the sound less harsh and robot-like. Other modules have their own speaker so the sound comes from the synthesiser itself, but this usually means you can't control the volume.

SAY THE WORD

Once plugged in, a number of synthesisers will immediately voice the letter or number every time a key is pressed, and on one version for the Spectrum all the keywords are pronounced as well. When speaking the letters, some give the phonetic sound—like 'huh'—others say the name of the letters—like 'aitch'. A few models even give you the choice.

However, in order to make the module say something useful you have to write a few lines of program. These can all be written in BASIC although sometimes it is necessary to do a bit of PEEKing and POKEing.

There are two different systems in common use for accessing the sound. With one method you simply write out the word using the special spellings of the allophones. Here's how you might say 'hello' on three different synthesisers (one each for the Spectrum, Commodores and Dragon):

```
10 LET s$="he(ll)(oo)":PAUSE 1
```

```
10 A$="H/E/LL/OO/"
20 SYS 41000
```

```
10 A$="HH1,EH,LL,OW,4"
20 SPEAK A$
```

You'll find the exact spelling of the allophones in each of the manuals. As you can see, it is all very simple and you soon learn which allophones make the sound you want. It becomes such second nature after a while that the only danger is you'll be spelling all words this way.

The other speech synthesisers use code numbers to refer to the allophones. You'll find that the numbers are common to several synthesisers as they are all based on the same chip, although the method of reading and

Microtip

Improving pronunciation

It is sometimes quite difficult to choose the correct set of allophones to make a word sound realistic.

The most common mistake is to follow the original spelling of the word rather than concentrate on the sounds from which it is made.

The box opposite shows that the spelling of a word gives no real guidance on the way it is pronounced. For example, *hall, trawl, bore, thought* and *poor* all use the same allophone 'or'. On the other hand, words that have similar spellings like *through, trough, borough* and *thought* are all pronounced quite differently.

It is worth programming the synthesiser to speak out its whole list of allophones—some manuals give a program to do this but it is not difficult to write your own. Then RUN the program a few times to get a better idea of what the sounds are like.

Another point to be aware of is that some synthesisers provide two or three versions of the same phoneme. For example, the 'b' sound in *bat* is slightly longer and more emphasized than the 'b' in *tab*. The longer sound is often represented by a double letter 'bb', so always check the position of the sound within a word before choosing its allophone.

Finally, intonation is most important as it can alter the whole meaning of the sentence. So it is worth buying a synthesiser that allows you to add stress or emphasis to a particular allophone.

using the numbers is different. The numbers for 'hello', for instance, are 27, 7, 45 and 53. The numbers are either entered in a line of DATA such as:

```
DATA 27,7,45,53
```

or entered as a string such as:

```
LET A$="27Ø74553"
```

The manuals for each type give plenty of examples showing exactly how the numbers are used.

Some systems allow you to add emphasis or intonation to the allophones to make the speech more lively and interesting, or to alter the actual sense of the words. This can be quite important. For instance, the sentence 'please listen to me' can mean three quite different things depending on whether you put the stress on the 'please', the 'listen' or the 'me'. One system uses capital letters to indicate which allophones are stressed, another uses certain number codes to alter the pitch of the sounds.

Once you've reached this stage you're well on your way to adding sounds to all sorts of programs. Speech synthesisers are simply great fun to play with and if you have the allophone type then they teach you a lot about language as well.

SPEECH SYNTHESISERS

Here is a list of the most common allophones used in speech synthesisers with some examples showing how they are pronounced. The actual spelling of the sounds in the synthesiser's user guide may vary, of course, but the sounds themselves should be the same as the ones given here.

And of course, individual pronunciations of words varies from person to person, and from area to area. The sounds given here are those which are recognized as Standard English.

VOWEL SOUNDS

Short vowels

a	cat shall carry
e	bet heavy
i	tin busy lynx
o	hot want
u	bug flood

Long vowels

ā	pear lane rein lair
ē	rear here veer seize
ī	ride buy dye sign die
ō	lone toad hoe crow
ū	duke new true

Other vowels

ar	card laugh pass
or	hall trawl bore thought
û	bird curl shirt
o͝o	could pull good
o͞o	blue food goose
ow	down bounce
oi	boil joy

CONSONANTS

Simple sounds

b	tab robber
d	day padding
f	full tough photo
g	gap guard
h	hide help
j	jape rage
k	chord rake car
l	loot pallor
m	mat hammer
n	no gnat knife
p	pip topping
r	rat wren
s	sit city scent
t	top thyme settle
v	vale live
w	won one tower
y	yet toying
z	zip has daze

Complex sounds

ch	hatch chum
dh	then father
gz	exact
ks	taxi packs
kw	queen quite
ng	ring zinc
ngg	tingle longer
sh	she chute sure
th	path three
wh	when why
zh	measure azure



A COMMODORE ASSEMBLER PROGRAM

Translating assembly language mnemonics into machine code hex can be a tedious business. But it is the sort of thing your Commodore 64 can easily do for you

The assembler for the Commodore 64 is a good deal shorter than the programs for the 48K Spectrum, Dragon and Tandy. This is because its 6510 processor—like the BBC's 6502—have less instructions than the Z80 or the 6809 chips.

The Commodore 64's assembler does not have to cope with the multitude of opcodes that the Z80 understands, nor does it have to work out the complex postbytes that you get with the Dragon and Tandy's 6809.

As well as being shorter, it is also faster. Even so, it may take some minutes to translate a long assembly language program into the corresponding machine code.

THE ASSEMBLER

```
10 POKE53280,Ø:POKE53281,Ø:
   PRINT"♥"
20 DIMK$(77),K1(77),K2(77):H$=
   "Ø123456789ABCDEF":R$=CHR$(13):
   QR$="■■■■■■■■■■"
30 DIM T$(2ØØ),RR(1ØØ),Z$(1ØØ)
40 P=Ø:FORII=1TO76:READK$(II),
   K1(II),K2(II):IFK$(II)<>"*"
   THENNEXT
50 DATAADC,1Ø5,1Ø1,AND,41,37,ASL,,
   6,BCC,,144,BCS,,176,BEQ,,24Ø,
   BIT,,36,BMI,,48
60 DATABRK,,,BYT,,-256
70 DATABNE,,2Ø8,BPL,,16,BVC,,8Ø,
   BVS,,112,CLC,24,,CLD,216,,CLI,
   88,,CLV,184,
80 DATACMP,2Ø1,197,CPX,224,228,CPY,
   192,196,DEC,,198,DEX,2Ø2,,DEY,
   136,,EOR,73,69
90 DATAINC,,23Ø,INX,232,,INY,2ØØ,,
   JMP,,68,JSR,,24
100 DATA,,,,,,,,,LDA,169,165,LDX,162,
   166,LDY
110 DATA16Ø,164,LSR,,7Ø,,,,,,,,,NOP,
   234,,,,,,,,,,,ORA,9,5,PHA,72,,
   PLA,1Ø4,,PLP
120 DATA4Ø,,PHP,8,,,,,,,,ROL,,38,ROR,,1Ø2
130 DATARTI,64,,RTS,96,,SBC,233,229,
   SEC,56,,SED,248,,SEI,12Ø,,STA,,
   133,STX,,134
140 DATASTY,,132,TAX,17Ø,,TAY,168,,
   TSX,186,,TXA,138,,TXS,154,,
   TYA,152,
150 DATA WOR,,-256,*,,
160 PRINT"♥"QR$"■□□□□□
   MENU■"
170 PRINTQR$"π1■□GET FROM TAPE"
   R$QR$"π2■□SAVE ON TAPE"
   R$QR$"π3■□ASSEMBLE"
180 PRINTQR$"π4■□EDIT LINE"
   R$QR$"π5■□DELETE LINE"
   R$QR$"π6■□LIST ON SCREEN"
190 PRINTQR$"■■ INPUT CHOICE?":
   POKE198,Ø
200 GETA$:IFVAL(A$)<1ORVAL(A$)>6
   THEN2ØØ
210 JJ=VAL(A$):PRINT"♥π"
220 ONJJGOSUB1Ø8Ø,11ØØ,25Ø,112Ø,
   12ØØ,123Ø:POKE198,Ø:PRINT
   "□■■■HIT ANY KEY"
230 GETA$:IFA$=""THEN23Ø
240 GOTO16Ø
250 KØ=Ø:K9=Ø:PS=Ø:PØ=Ø
260 PS=PS+1:IFPS<=3THENK=KØ:
   P=PØ:PRINT"■■□PASS□:■"
   PS:GOTO28Ø
270 PØ=P:RETURN
280 GOSUB98Ø
290 GOSUB91Ø:OP$=I$:IFLEFT$(OP$,1)
   ="*"ANDPS=3THENPRINTOP$
300 IFLEFT$(OP$,1)="*"THEN28Ø
310 IFOP$="END"ANDPS=3THENPRINT
   "■ END LAST ADDR";P-1
320 IFOP$="END"THEN26Ø
330 IFOP$<>"ORG"ANDOP$<>"P%="
   THEN37Ø
340 GOSUB91Ø:S=Ø:IFLEFT$(I$,1)=
   "*"THENS=P:I$=RIGHT$(I$,LEN
   (I$)-1)
350 P=VAL(I$)+S:IFPS=3THENPRINT
   "■□□□□ORG"P
360 GOTO28Ø
370 IFP=ØTHENPRINT"♥YOU FORGOT
   ORG":RETURN
380 IFOP$<>"$P%="ANDOP$<>
   "TXT"THEN42Ø
390 GOSUB91Ø:IFPS=3THENPRINT
   "□□□□TXT "I$TAB(2Ø);
400 FORJ=2TOLEN(I$):BY=ASC(MID$
   (I$+I$,J,1)):IFJ>=LEN(I$)
   THENBY=13
410 GOSUB87Ø:NEXTJ:GOTO28Ø
420 IFLEN(OP$)<>3THENFORI=1TO1:
   GOTO44Ø
430 FORI=1+3*(ASC(OP$)-65)TO76:
   IFOP$=K$(I)THEN49Ø
440 NEXT:IFPS=3THENPRINTOP$
450 IFLEFT$(I$,1)="."THENI$=RIGHT$
   (I$,LEN(I$)-1)
460 GOSUB1Ø1Ø:RR(Q2)=P:IFI$=""
   THEN29Ø
470 IFPS=3THENPRINT"(THIS LINE NOT
   RECOGNIZED)"
480 GOTO28Ø
490 AD$="#Ø":R=Ø:IFK2(I)<>ØTHEN
   GOSUB91Ø:AD$=I$:OP=K2(I)
500 IFPS=3THENPRINT"□"OP$;:IFK2(I)
   <>ØTHENPRINT"□"AD$;
510 IFLEFT$(AD$,1)="#"THENAD$=
   RIGHT$(AD$,LEN(AD$)-1):
   OP=K1(I)
520 IFRIGHT$(AD$,1)<>")"THEN53Ø
525 OP=OP-2Ø:AD$=MID$(AD$,2,
   LEN(AD$)-2):IFOP=48THENOP=1ØØ
530 IFRIGHT$(AD$,2)=",X"THENOP=OP
   +16:AD$=LEFT$(AD$,LEN(AD$)-2)
540 IFAD$="A"THENAD$="Ø":OP=OP
   +4
550 IFRIGHT$(AD$,2)<>",Y"THEN58Ø
560 OP=OP+16:AD$=LEFT$(AD$,LEN
   (AD$)-2):IFRIGHT$(OP$,1)<>"X"
   THENOP=OP-4:R=65536
570 IFRIGHT$(AD$,1)=")"THENAD$=
   MID$(AD$,2,LEN(AD$)-2):R=R-65536
580 S=1
590 IFAD$=""THEN77Ø
600 X$=LEFT$(AD$,1):BD$=RIGHT$
   (AD$,LEN(AD$)-1)
610 IFX$="*"THENR=R+P*S:AD$=BD$:
   GOTO58Ø
620 IFX$="+"THENAD$=BD$:GOTO59Ø
630 IFX$="-"THENAD$=BD$:S=-S:
   GOTO59Ø
640 Q=Ø
650 IF(X$<>"$"ANDX$<>"&")ORBD$
   <"Ø"ORBD$>="G"THEN69Ø
660 FORQ2=ØTO15:IFLEFT$(BD$,1)
```

- TRANSLATING ASSEMBLY LANGUAGE INTO HEX MACHINE CODE
- WORKING OUT JUMPS AND BRANCHES
- COPING WITH LABELS
- **POKE**ING IN THE HEXADECIMAL MACHINE CODE




```
   < >MID$(H$,Q2+1,1)THEN68Ø
67Ø Q=Q*16+Q2:BD$=RIGHT$(BD$,
   LEN(BD$)−1):GOTO66Ø
68Ø NEXT:R=R+Q*S:AD$=BD$:GOTO58Ø
69Ø IFX$<"A"ORX$>"Z"THEN72Ø
7ØØ I$=AD$:GOSUB1Ø1Ø:IFI$< >""THEN
   GOSUB1Ø5Ø
71Ø R=R+RR(Q2)*S:AD$=I$:GOTO58Ø
72Ø IFX$<"Ø"ORX$>=":"THENR=Ø:
   GOTO76Ø
73Ø IFAD$<"Ø"ORAD$>=":"THEN75Ø
74Ø Q=Q*1Ø+ASC(AD$)−48:AD$=
   RIGHT$(AD$,LEN(AD$)−1):GOTO73Ø
75Ø R=R+S*Q:GOTO58Ø
76Ø IFPS=3THENPRINT"(ADDRESS NOT
   UNDERSTOOD)";
77Ø IFLEFT$(OP$,1)="B"AND(Ø=(OP
   AND4)ANDOP>Ø)THENR=(R−P−2)
   AND255
78Ø IFLEFT$(OP$,1)="J"ORLEFT$(OP$,1)
   ="W"THENR=R+65536
79Ø IFR>255THENOP=OP+8
8ØØ K2=K2(I):IF(OPAND15)=1Ø
   THENK2=Ø
81Ø IFPS=3THENPRINTTAB(16);:BY=
   P/256:GOSUB89Ø:BY=P−32768:
   GOSUB89Ø:GOSUB85Ø
82Ø IFOP>=ØTHENBY=OP/256:GOSUB
   87Ø:BY=OP:GOSUB88Ø
83Ø IFK2=ØTHEN28Ø
84Ø GOSUB85Ø:BY=R−256*INT(R/256):
   GOSUB88Ø:BY=R/256:GOSUB87Ø:
   GOTO28Ø
85Ø IFPS=3THENPRINT"□";
86Ø RETURN
87Ø IFINT(BY)<=ØTHENRETURN
88Ø P=P+1:IFPS=3THENPOKEP−1,
   BYAND255
89Ø BY=BYAND255:IFPS=3THENPRINT
   MID$(H$,BY/16+1,1)MID$
   (H$,(BYAND15)+1,1);
9ØØ RETURN
91Ø IFK>NTHENI$="END":RETURN
92Ø K1=K9+1:IFK9>=LEN(T$(K))THEN
   I$="/MISSING/":RETURN
93Ø K9=K1:IFMID$(T$(K),K1,1)=
   "□"THEN92Ø
94Ø IFK9>LEN(T$(K))THENI$=MID$
   (T$(K),K1,K9−K1):RETURN
95Ø IFMID$(T$(K),K9,1)< >"□"THEN
   K9=K9+1:GOTO94Ø
96Ø I$=MID$(T$(K),K1,K9−K1)
97Ø RETURN
98Ø IFK9<=LEN(T$(K))ANDPS=3THEN
   PRINTRIGHT$(T$(K),LEN(T$(K))
   −K9+1);
99Ø K=K+1:K9=Ø:IFPS=3THENPRINT
1ØØØ RETURN
1Ø1Ø X$=""
1Ø2Ø IFI$<"A"ORI$>="["THEN1Ø4Ø
1Ø3Ø X$=X$+LEFT$(I$,1):I$=RIGHT$
   (I$,LEN(I$)−1):GOTO1Ø2Ø
1Ø4Ø IFI$< >""THENRETURN
1Ø5Ø FORQ2=1TOVV:IFX$=Z$(Q2)
   THEN1Ø7Ø
1Ø6Ø NEXT:VV=VV+1:Z$(VV)=X$:
   Q2=VV:RR(VV)=32768
1Ø7Ø RETURN
1Ø8Ø OPEN1,1,Ø,"ASM"
1Ø9Ø INPUT#1,N:FORJ=1TON:INPUT#1,
   T$(J):NEXT:CLOSE1:RETURN
11ØØ OPEN1,1,1,"ASM"
111Ø PRINT#1,N:FORJ=1TON:PRINT#1,
```

```
    CHR$(34)+T$(J)+CHR$(34):
    NEXT:CLOSE1:RETURN
1120 K=0:INPUT"♡π ENTER LINE
    NUMBER✝";K:PRINT"♡"
1130 POKE198,5:POKE631,144:POKE632,
    34:POKE633,34:POKE634,20:
    POKE635,30
1140 IP$="":PRINT"π"K;:INPUTIP$:
    IFIP$=""THENRETURN
1150 K2=K/10:IFK2>NTHENK2=N+1:
    N=N+1
1160 IFK2<.1THENK2=.1
1170 IFK2=INT(K2)THEN1190
1180 K2=INT(K2)+1:FORK3=NTOK2
    STEP-1:T$(K3+1)=T$(K3):NEXT:
    N=N+1
1190 T$(K2)=IP$:K=K+10:GOTO1130
1200 K=0:INPUT"♡π ENTER LINE
    NUMBER✝";K:K2=K/10
1210 IFK2>NORK2<1ORK2>INT(K2)THEN
    RETURN
1220 FORK3=K2TON:T$(K3)=T$(K3+1):
    NEXT:N=N-1:RETURN
1230 IFN=0THENRETURN
1240 K=0:K2=0:INPUT"♡π ENTER
    FIRST, LAST LINE NUMBERS✝";
    K,K2:K1=K/10:K2=K2/10
1250 IFK2>NTHENK2=N
1260 IFK1<1THENK1=1
1270 PRINT"♡":FORK3=K1TOK2:PRINT
    "π"K3*10"✝ □"T$(K3):NEXT:
    RETURN
```

HOW IT WORKS

Once you have keyed in the program, type RUN to initialize it. Usually you will be putting your machine code program into the 4K area left empty for it from C000 to CFFF hex, 49,152 to 53,247 decimal. If you want to put your machine code into the BASIC area you must clear a space for it by POKEing memory locations 52 and 56 first (see page 278).

When the program has been RUN, the menu will appear. To enter your assembly language program you should select option 4, which says 'edit line'. The assembler uses BASIC line numbers in multiples of ten.

Your first line—number 10, say—should carry the origin or start address of the machine code program. Usually this takes the form:

```
10 ORG 49152
```

This puts the machine code translation of your assembly language program into the area of RAM left spare for the purpose. If you have cleared an area of the BASIC RAM for your machine code program the origin should be the beginning of that.

After that, line numbers in multiples of 10 appear automatically every time you press RETURN. A single mnemonic, along with its associated address and data, should occupy each line. If you want to insert a line, use an intermediate line number—the assembler will automatically round up. Labels at the beginning of a line—showing where the program should jump to—should start with a full stop and a space. The name of the label itself will do after a branch or jump instruction. Labels must be more than one letter or anything else that cannot be confused with a mnemonic. Otherwise standard 6510 mnemonics are used.

If you want to skip a line—perhaps because you want to fill it in later—use a *. While you are entering a line of machine code you can edit it on screen in the normal way like a line of BASIC. When you push RETURN you automatically move onto the next line. But if you press RETURN again before you have typed anything on the line, the program will return you to the menu.

The menu gives you an option to list the program. If you press another key you'll go back to the menu again. And from there you can choose the option to delete a line, or you can overwrite it in the edit mode.

Once you are satisfied that you have written the program correctly, select the assemble option. The Commodore 64 will then list your assembly language program—minus BASIC line numbers—along with the machine code.

The assembler itself can be SAVEd and LOADed just like any other BASIC program (see page 22). The assembly mnemonics—or source code—can be saved on tape by using the save option on the menu. And it can be loaded back into the computer using the 'get from tape' option.

The machine code program itself—or object code—can be saved by entering the Commodore machine code monitor (see page 280), and using the save option on that.

To run the machine you have to BREAK out of the assembler program and use:

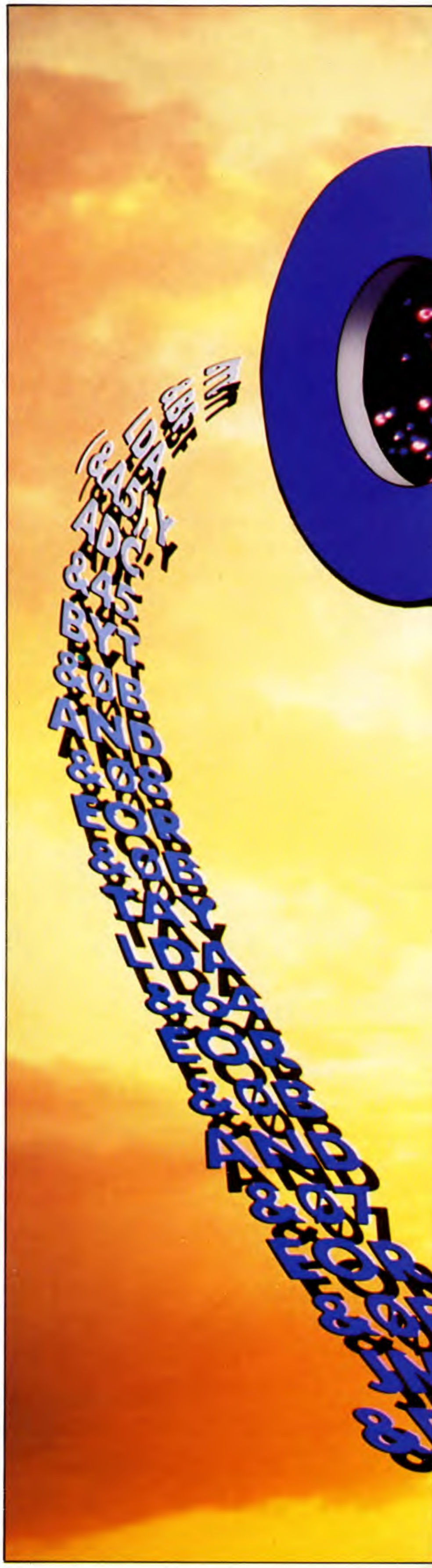
```
SYS 49152
```

TESTING

To test your assembler try keying in the assembly language right-scrolling program given on page 326. Whether you hand assemble that program or feed it into your assembler, the resulting machine code should read:

```
A9 00 85 FB A9 04 85 FC A9 00 85 FE A0 27
B1 FB 85 FD A0 26 B1 FB C8 91 FB 88 88 C0
FF D0 F5 A0 00 A5 FD 91 FB A5 FB 69 27 85
FB 90 02 E6 FC E6 FE A6 FE E0 19 D0 D5 60
```

INPUT is not carrying a Vic 20 assembler because it is not possible to write one in





BASIC that could be used for the programs that will be covered in following chapters. If you are a Vic owner and are interested in exploring machine code further, it is suggested that you buy one of the machine code assemblers available commercially.

If you have one, try it out on the right-scrolling program on page 326. Whether you use an assembler or hand assemble that program you should get:

```
A9 00 85 FB A9 1F 85 FC A9 00 85 FE A0 15
B1 FB 85 FD A0 14 B1 FB C8 91 FB 88 88 C0
FF D0 F5 A0 00 A5 FD 91 FB A5 FB 69 15 85
FB 90 02 E6 FC A6 FE E6 FE E0 16 D0 D5 60
```

Q+A

What do I do if a long program—like my assembler—does not work when I key it in?

Even the most experienced programmer has trouble when keying in a long program. It is almost impossible to key in a program of any length without making an error, no matter how careful you are. The question is: how to find the bugs?

Many of the problems you will come up against will give you an error message which will lead you to the bug (see pages 334 to 338). But in long programs the computer may go back and forth, around loops and over jumps many times and an error message that simply points you to a line number may not be much help. A line may work 20 times, say, but on the 21st time it may falter because a variable has been set to a troublesome value. Or, perhaps an incorrect IF . . . THEN statement comes into play.

What you need is a trace program that will print out the number of every line as it is being executed. *INPUT* will be publishing one for the Commodore 64—and the Vic 20—in a forthcoming issue.

So do not despair if your assembler does not RUN first time. Check it carefully for all the common errors. If you get an 'OUT OF DATA' error message, for example, check your DATA statements closely. It is very difficult to check so much DATA, but if you have left out one number—or even a comma—the program will not RUN.

But if you are still having trouble, wait for the Commodore trace program. It is a powerful tool for tracking errors.

THE POWERS THAT BE

Here are some more maths functions that have plenty of practical applications, from finding out the area of your floor to working out the speed of a falling body

The 'power' function is a very often overlooked maths function which has a surprisingly wide variety of uses, especially if you want to work out areas or volumes in your programs.

The power function on the computer looks like this: ↑. It is put in between two numbers, like 2↑3, for example, so you might hear someone say: 'Two to the power three.'

Although this may look confusing, it is actually a simple idea. A number 'to the power of' another number is simply the first number multiplied by itself a number of times. The second number tells you how many times.

So, to return to the example above, two to the power three is two multiplied by two three times over. It looks like this:

2↑3=2*2*2=8

This is the same for any pair of numbers. So:

2↑5=2*2*2*2*2=32
5↑4=5*5*5*5=625

You can also write the numbers like this: 2^3; 2^5; and 5^4, which is the more conventional mathematical notation but this is not understood by the computer.

SQUARES AND CUBES

There are special names for the powers of two and three. Any number to the power of two is 'squared'. And any number to the power of three is usually called the number 'cubed'.

There are actually good reasons for the names attached to these two particular powers.

If you are trying to measure the area of a rectangle (whether it is your lawn or your front room carpet or any other rectangular shape) you measure the length of the rectangle and then measure its width. To find the area, all you do is multiply the two measurements together.

The result of this calculation will be a number of 'square' units. The reason that the area is measured in, say, square metres is because what you are in fact doing is measuring how many squares with both sides one metre long will fill your area.

Similarly a number to the power of two is called a squared number—a square area is represented by a multiplication of the basic unit by itself.

In the same way as the power of two is named after area measurement, the power of three is named after volume measurement. To find the volume of a cube, you multiply the cube's width by its depth by its height. So there are three multiplications of the basic unit needed to find the volume (two for the area, plus one more for the height), and a number to the power of three is that number 'cubed'.

MORE POWERS

Beyond this point, it is not possible to build a model for what the power means. If it was possible to have an object with four dimensions, then its volume would be measured in units to the power of four – but this is obviously absurd. However, there are plenty of calculations when these larger powers can prove useful for other reasons.

For example, suppose you want to know the probability of a dice coming up a six. This is easy—every face has an equal chance of showing, so the chance of it being any particular number is $\frac{1}{6}$. But now suppose that you want to know the chance of getting two sixes in a row. There is a one in six chance of the first being a six, and *if* this happens, there is also a one in six chance of the next one being a six. So the overall probability is $\frac{1}{6}$ *times* $\frac{1}{6}$. In other words it is $\frac{1}{6}$ *squared*. And this holds true for as long as you want to go on—the chance of seven dice all coming up the same is $\frac{1}{6}*\frac{1}{6}*\frac{1}{6}*\frac{1}{6}*\frac{1}{6}*\frac{1}{6}*\frac{1}{6}$ $=(\frac{1}{6})$↑7—a very small number.

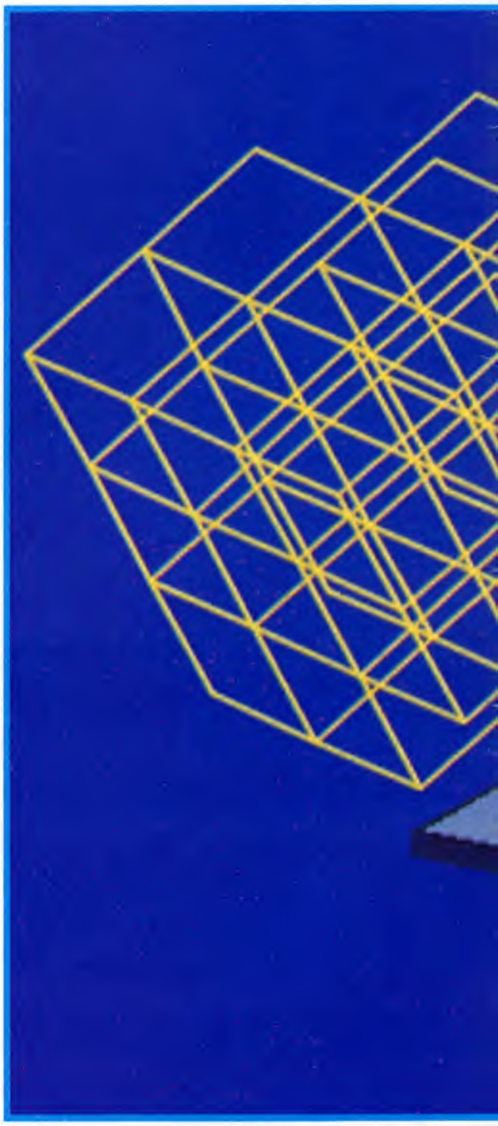
You have already come across at least one example of numbers to these higher powers—and that is when converting to binary. Each binary digit represents a power of the number two. So in the binary number 11111111, the first number on the left (bit Ø) represents a 1, the second (bit 1) is a 2, the third (bit 2) is a 4, and so on; 8, 16, 32, 64, 128.

Look at these a little more closely: 4 is 2*2, or 2 to the power 2; 8 is 2*2*2, or 2 to the power 3; 16 is 2*2*2*2, and so on:

```
2↑2=2*2            =4
2↑3=2*2*2          =8
2↑4=2*2*2*2        =16
2↑5=2*2*2*2*2      =32
2↑6=2*2*2*2*2*2    =64
2↑7=2*2*2*2*2*2*2=128
```

Two values are missing from this table. Although it is easy to see what they must be, the answer is perhaps a little surprising. The first of these is 2↑1. Obviously, from the binary analogy, this must be 2. In fact, by extension, if 2↑2 is 2 multiplied by itself once, 2↑1 must be 2 multiplied by itself one less time—in other words 2 is *not* multiplied by itself at all, so it stays as two.

The value below this is perhaps even more surprising—2↑Ø. You might expect this to be Ø, but in fact from the table above it is clearly



- THE POWER FUNCTION
- SQUARING AND CUBING NUMBERS
- RAISING NUMBERS TO HIGHER POWERS
- FINDING THE AREA OF A SQUARE
- SQUARE ROOTS
- USING THE SQR FUNCTION IN A PROGRAM

1. This is because it must be 2 multiplied by itself 1 less time than 2↑1. Since 2↑1 is the number 2 itself, the only way in which you can multiply this by 2 one less time is actually to *divide* 2 by 2. And any number divided by itself is 1.

To check these, or other power values you want to experiment with, type in:

```
PRINT 2↑X
```

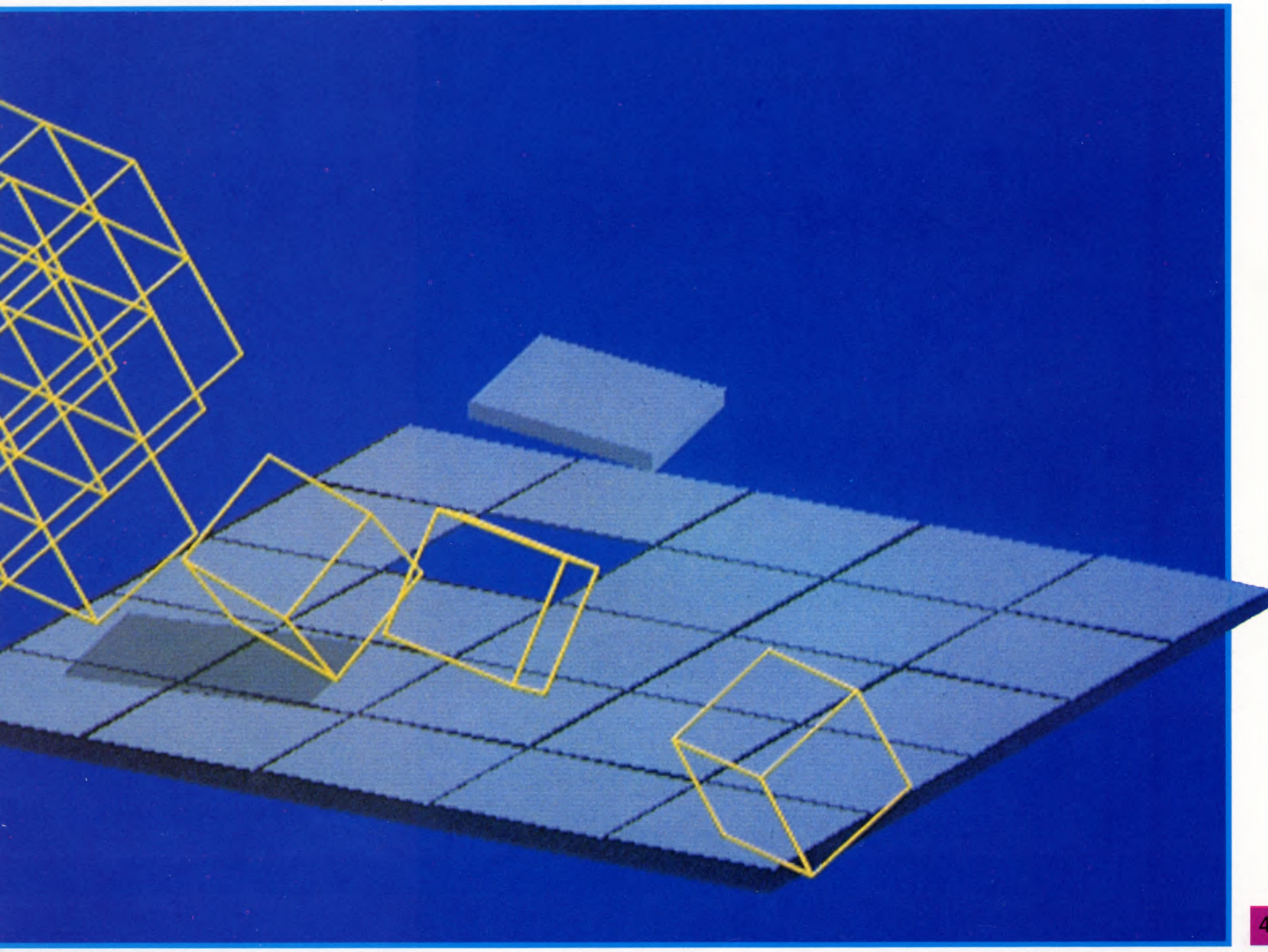
where X is the value you want to check, and press ENTER or RETURN. You can try any value you like (although large ones may give overflow errors). You may come across some rather interesting results—for example, what happens if you type in a fractional value for X—.5, say? We will return to this later, but first a closer look at the widely used square power.

DRAWING SQUARES

The program below takes a series of numbers, and uses each in turn to draw a series of squares on the screen on your television/monitor, and shows both the length of one side of each square, and its area. It then allows you to pick out any two squares and to compare their areas. It will also show you how rapidly the square grows compared to the base number, and you should find that the whole idea of powers makes much more sense.

S

```
10 CLS
20 PRINT AT 2,0;"Length";AT 3,0;
   " of";AT 4,0;"side"
30 PRINT AT 2,25;"Area";AT 3,25;
```



```
   " of";AT 4,25;"square"
40 LET z=0:PLOT 55,23
50 FOR n=1 TO 13
60 LET m=n: IF m>7 THEN
   LET m=m-8
70 DRAW 0,n*8
80 DRAW PAPER m;n*8,0
90 DRAW PAPER m;0,-(n*8)
100 DRAW PAPER 7;-(n*8),0
110 PRINT AT 6+(13-n),0;n;AT
   6+(13-n),25;n*n
120 PAUSE 10:NEXT n
140 INPUT "would you like to compare any
   areas? (y/n)";a$
150 IF a$<>"y" AND a$<>"n" THEN
   GOTO 140
160 IF a$="n" THEN GOTO 300
170 INPUT "which is the first square whose area
   you want to compare?(1-13)";a: IF a<1
   OR a>13 THEN GOTO 170
180 INPUT "and which is the
   second?(1-13)";b: IF b<1 OR b>13
   THEN GOTO 180
190 LET x=INT (a): LET y=INT (b)
200 IF x>y THEN GOTO 280
210 CLS : PRINT "The first area fits into the
   second one□";y∧2/x∧2; "□times."
220 PLOT 20,0: DRAW x*8,0: DRAW 0,x*8:
   DRAW -x*8,0: DRAW 0,-x*8: DRAW
   y*8,0: DRAW 0,y*8: DRAW -y*8,0:
   DRAW 0,-y*8
230 PRINT "do you want to compare any more
   areas?(y/n)"
240 INPUT a$
250 IF a$<>"y" AND a$<>"n" THEN
   GOTO 240
260 IF a$="n" THEN LET z=1: GOTO 300
270 GOTO 170
280 CLS : PRINT "The first area
   is□";x∧2/y∧2;"□times larger than the
   second."
290 GOTO 220
300 IF z=1 THEN CLS : LET z=0
310 PRINT INK 2; FLASH 1;AT 0,0;"Press any
   key to see this again"
320 PAUSE 0
330 IF INKEY$="n" THEN STOP
340 RUN
```

```
1 POKE 53280,1:POKE 53281,1
2 PRINT "■":GOSUB 8000
10 FOR Z=20 TO 1 STEP -1
20 FOR A=1 TO Z:POKE 1912-Z*40+A,
   99:POKE 56184-Z*40+A,3
25 POKE 1913-A*40+Z,101:POKE
   56185-A*40+Z,3
26 POKE 1912+Z,99:POKE 56184+Z,3
30 NEXT A:PRINT "■"Z;TAB(8)"■□"
   TAB(32)"■"Z*Z:NEXT Z:H=0
35 PRINT "■■■■■■■■■■
   ■■■■■■■■■■■■■
   ■■WOULD YOU LIKE TO COMPARE
   ANY AREAS? ■Y■/■N";
40 POKE 198,0
100 GET A$:IF A$<>"Y" AND
   A$<>"N" THEN 100
105 PRINT "■";:FOR Z=1 TO 40:PRINT
   "□";:NEXT Z
106 IF H=1 THEN C=3:Z=X:GOSUB
   600:Z=Y:GOSUB 600
110 H=1:IF A$="N" THEN GOSUB
   8000:GOTO 35
115 GOSUB 500
120 INPUT "■■ENTER FIRST SQUARE.
   (1-20)■";A$:X=VAL(A$):
   IF X<1 OR X>20 THEN 115
125 GOSUB 500
130 INPUT "■■ENTER SECOND SQUARE.
   (1-20)■";B$:Y=VAL(B$):
   IF Y<1 OR Y>20 THEN 125
140 C=2:Z=X:GOSUB 600:Z=Y:GOSUB
   600
150 IF X>Y THEN 200
160 GOSUB 500:PRINT "■■THE FIRST
   AREA FITS INTO THE SECOND
   ONE"Y↑2/X↑2"TIMES."
170 GOTO 35
200 GOSUB 500
210 PRINT"■■THE FIRST AREA IS"
   X↑2/Y↑2"TIMES":PRINT
   "■LARGER THAN THE SECOND."
220 GOTO 35
499 GOTO 499
500 PRINT"■";:FOR Z=1TO80:
   PRINT"□";:NEXT Z:RETURN
600 FOR A=1 TO Z:POKE 1912-Z*40+A,
   99:POKE 56184-Z*40+A,C
610 POKE 1913-A*40+Z,101:POKE
   56185-A*40+Z,C:NEXT A:RETURN
8000 GOSUB 500
8010 PRINT "■■■LENGTH OF SIDE
   □□□□□□□□□□□□□□
   AREA OF SQUARE":RETURN
```

```
1 POKE 36879,25
2 PRINT "■":GOSUB 8000
10 FOR Z=9 TO 1 STEP -1
20 FOR A=1 TO Z:POKE 8080-Z*22+A,
   99:POKE 38800-Z*22+A,3
25 POKE 8081-A*22+Z,101:POKE
   38801-A*22+Z,3
26 POKE 8080+Z,99:POKE 38800+Z,3
30 NEXT A:PRINT "■"Z;TAB(4)"■□"
   TAB(17)"■"Z*Z:NEXT Z:H=0
35 PRINT "■■■■■■■■■■
   ■■■■■■■■■■■■
   WOULD YOU LIKE TO COMPARE ANY
   AREAS? Y/N";
40 POKE 198,0
100 GET A$:IF A$<>"Y" AND
```





```
   A$<>"N" THEN 100
105 PRINT "◘◘";:FOR Z=1 TO 44:
   PRINT "□";: NEXT Z
106 IF H=1 THEN C=3:Z=X:GOSUB 600:
   Z=Y:GOSUB 600
110 H=1:IF A$="N" THEN GOSUB 8000:
   GOTO 35
115 GOSUB 500
120 PRINT"▤◧ENTER FIRST SQUARE.
   (1-9)";:GETA$:X=VAL(A$):
   IF X<1 OR X>9 THEN 120
125 GOSUB 500
130 PRINT"▤◧ENTER SECOND SQUARE.
   (1-9)";:GETB$:Y=VAL(B$):
   IF Y<1 OR Y>9 THEN 130
135 X=INT(X): Y=INT(Y)
140 C=2:Z=X:GOSUB 600:Z=Y:
   GOSUB 600
150 IF X>Y THEN 200
160 GOSUB 500:PRINT "▤◧THE FIRST
   AREA FITS□□□INTO THE SECOND
   ONE□□□"Y↑2/X↑2"TIMES."
170 GOTO 35
200 GOSUB 500
210 PRINT"▤◧THE FIRST AREA IS□□
   □□□"X↑2/Y↑2"TIMES":PRINT
   "◘◘◙LARGER THAN THE FIRST."
220 GOTO 35
500 PRINT"▤◙◧";:FOR Z=1TO88:
   PRINT"□";:NEXT Z:RETURN
600 FOR A=1 TO Z:POKE 8080-Z*22+A,
   99:POKE 38800-Z*22+A,C
610 POKE 8081-A*22+Z,101:POKE 38801
   -A*22+Z,C:NEXT A:RETURN
8000 GOSUB 500
8010 PRINT "▤◙◧LENGTH OF□□□
   □□□AREA OF";
8020 PRINT "SIDE□□□□□□□□□□
   □□□SQUARE◙◙◙◙◙◙":
   H=0:RETURN
```

```
10 MODE1
20 @%=&5
30 PROCLABEL
40 PROCDRAW
50 PROCCOMPARE
60 GOTO 30
70 DEF PROCDRAW
80 FOR T=1 TO 20
90 MOVE300,64:DRAW300,T*32+64:
   DRAWT*32+300,T*32+64:DRAW
   T*32+300,64:DRAW300,64
100 PRINTTAB(1,30-T)T;TAB (32,
   30-T)T*T
110 NEXT
120 ENDPROC
130 DEFPROCLABEL
140 CLS
150 PRINTTAB(2,6)"LENGTH"TAB
   (4,7)"OF"TAB(2,8)"TABLE"
160 PRINTTAB(34,6)"AREA"TAB
   (35,7)"OF"TAB(33,8)"SQUARE"
170 ENDPROC
180 DEF PROCCOMPARE
190 INPUTTAB(0,2)"DO YOU WANT
   TO COMPARE ANY AREAS
   (Y/N) ",A$
200 IF A$<>"Y" AND A$<>
   "N" THEN 190
210 IF A$="N" THEN ENDPROC
220 CLS
230 INPUTTAB(0,10)"ENTER THE
   LENGTH OF THE FIRST SQUARE",X
240 X=INT(X):IF X<1 OR X>20 THEN
   GOTO 220
250 CLS:INPUTTAB(0,10)"ENTER THE
   LENGTH OF THE SECOND SQUARE",Y
260 Y=INT(Y):IF Y<1 OR Y>20 THEN
   GOTO 250
270 CLS
```

```
28Ø MOVE3ØØ,64:DRAW3ØØ,X*32+64:
    DRAWX*32+3ØØ,X*32+64:DRAW
    X*32+3ØØ,64:DRAW3ØØ,64
29Ø MOVE3ØØ,64:DRAW3ØØ,Y*32+64:
    DRAWY*32+3ØØ,Y*32+64:DRAW
    Y*32+3ØØ,64:DRAW3ØØ,64
3ØØ IF X>Y THEN GOTO 35Ø
31Ø PRINTTAB(Ø,3)"THE FIRST AREA FITS
    INTO THE SECOND
    ONE□";Y∧2/X∧2;"□TIMES"
32Ø A=GET
33Ø CLS
34Ø GOTO19Ø
35Ø PRINTTAB(Ø,3)"THE FIRST AREA
    IS□";X∧2/Y∧2;'"TIMES LARGER THAN
    THE SECOND."
36Ø GOTO 32Ø
```

```
1Ø PMODE4,1:PCLS:SCREEN1,1
2Ø FOR N=17 TO 1 STEP -1
3Ø LINE(2Ø,171)-(2Ø+8*N,171-
    8*N),PSET,B
4Ø NEXT
5Ø CLS
6Ø IF INKEY$="" THEN 6Ø
7Ø PRINT" DO YOU WISH TO
    COMPARE ANY□□□□□□□□□
    AREAS (Y/N) ?"
8Ø A$=INKEY$:IF A$<>"Y" AND
    A$<>"N" THEN 8Ø
9Ø IF A$="N" THEN 22Ø
1ØØ PRINT:INPUT"□WHICH IS THE FIRST
    SQUARE WHOSE AREA YOU WANT TO
    COMPARE (1-17)";A:A=INT(A):IF A<1
    OR A>17 THEN 1ØØ
11Ø PRINT:INPUT"AND WHICH IS THE
    SECOND (1-17)□";B:B=INT(B):IF
    B<1 OR B>17 THEN 11Ø
12Ø IF A>B THEN 2ØØ
13Ø CLS:PRINT"□THE FIRST AREA FITS INTO
    THE□□□□SECOND
    ONE";B↑2/A↑2;"TIMES."
14Ø FOR K=1 TO 2ØØØ:NEXT
15Ø PCLS:SCREEN 1,1
16Ø LINE(2Ø,171)-(2Ø+8*A,171-8*A),PSET,
    B:LINE(2Ø,171)-(2Ø+8*B,171-
    8*B),PSET,B
17Ø IF INKEY$="" THEN 17Ø
18Ø CLS:PRINT"□DO YOU WANT TO
    COMPARE ANY MORE AREAS (Y/N) ?"
19Ø GOTO 8Ø
2ØØ CLS:PRINT"□THE FIRST AREA IS";
    A↑2/B↑2:PRINT@32,"TIMES□□
    □□□□LARGER THAN THE FIRST."
21Ø GOTO 14Ø
22Ø CLS
23Ø PRINT@33,"PRESS ANY KEY TO START
    AGAIN."
24Ø IF INKEY$="" THEN 24Ø
25Ø GOTO 1Ø
```

The Vic 20 version of this program will not look as if the objects drawn by the computer are actually squares. This is because the Vic's pixels are not square shaped, but are wider than they are high, which makes them look like rectangles.

The Dragon cannot PRINT on its graphics screen, and so overcomes the problem of showing you the squares and their measurements by changing from one screen to another whenever necessary.

Each of the computers, except the Commodores, draws the squares one at a time, starting with the smallest. The Commodore machines have no DRAW command accessible in their standard BASIC, and so have to use ROM graphics which are PRINTed. The large squares which have their lines at, or near, the top must be PRINTed first. This is because PRINT commands always operate at the beginning of the next highest line on the screen, unless you program them not to. So the first PRINT command PRINTs on the top line of the screen, the next on the next line, and so on, with the result that the largest squares are displayed first.

Once the squares have all been drawn or PRINTed on the screen the Acorn programs PRINT the measurements, which have already been done by the Commodore and Spectrum. The Dragon waits for you to press a key, and then goes straight to the next routine as it cannot PRINT on the text screen.

This routine allows you to compare the areas of two squares. It waits for you to INPUT a number, this being the number of the first square you want to compare, and checks to make sure that it is a 'legal' number. In other words if the number is larger or smaller than the number of squares then it will not be accepted. If you INPUT a decimal fraction, the computer converts it into a whole number using the INT functions.

Once you have INPUTted two legal numbers, the computer sees which is larger. This is so that you do not see a message like this: 'the first area fits into the second one Ø.xxx times' (which would happen whenever the first number is larger than the second).

The computer then squares each number (i.e. it multiplies each number by itself) and divides the larger number by the smaller. The message telling you which is larger, and how much larger it is, is then PRINTed on the screen. You are then offered the chance to compare two areas again, if you want to, or to see the whole program again.

SQUARE ROOTS

The power of two, or the square function, is probably the most frequently used of the





powers. But you will quite often want to work out the square in reverse, that is, to find the original number of a square that you know. For example, if you know that the area of a square is, say, 81, you may want to find out how long each side is.

With the number 81, this is not too difficult: nine times nine is 81, so the length of each side must be 9. But had the area of the square been a more obscure number, like 127, it would be much harder to work out how long each side is. The computers have a function to help you with this: the square root function.

Type PRINT SQR(81) into your computer and press RETURN or ENTER. You should see the number 9 appear on your screen. (If you want to find out what the square root of 127 is, as a matter of interest, do the same thing but replace 81 with 127.)

The computer has a special SQR command because the square root is so widely used, but in fact this is not the only way you can work out a square root—you can also use the ordinary power function. If you have experimented with fractions in this function, you may have already discovered that 2↑.5—two to the power $\frac{1}{2}$—has the same effect as SQR(2). To verify this, try PRINT SQR(81), then PRINT 81↑.5. The results may not be exactly the same but they should be close.

You can perform a similar operation with $\frac{1}{3}$, which works out the reverse of the cube—the cube root—and similarly for any other power. Although these have their uses, the square root is the most widely used of all.

You can use the SQR command for much more than just finding out the length of a side of a square whose area you know. Quite a few mathematical equations have square numbers and square roots in them, and where this is the case you can use the SQR function to work out the answer on your computer.

For example, the next program uses an equation to work out how long it will take an object to hit the ground if it falls off a cliff, or some other great height (ignoring air resistance). It also tells you how fast the object will be travelling when it hits the ground. The reason this is linked to squares and square roots is that anything which falls under the influence of gravity falls faster and faster, the longer it has been falling (ignoring air resistance). In fact, it is linked to the square of the time. But before looking at this in detail, try out the program itself:

```
10 CLS
20 INPUT "ENTER FALLING DISTANCE
   (METRES)",D
30 IF D<0 THEN GOTO 20
40 LET T=SQR ((2*D)/9.81)
50 LET V=SQR (2*D*9.81)
60 LET T=INT (T*100)/100
70 LET V=INT (V*100)/100
80 PRINT INVERSE 1'"TIME TAKEN TO
   REACH GROUND:"; INVERSE
   0'T;"□SECONDS"
90 PRINT INVERSE 1'"MAXIMUM VELOCITY
   REACHED:"; INVERSE 0'V;"□METRES
   PER SECOND"
100 PRINT "(";INT (2.25*V+.5);"□MPH)"
200 PRINT ''"PRESS ANY KEY TO RUN
   AGAIN"
210 IF INKEY$="" THEN GOTO 210
220 GOTO 10
```

```
10 PRINT "■"
20 INPUT "ENTER FALLING DISTANCE
   (METRES)";D
30 IF D<0 THEN 20
40 T=SQR ((2*D)/9.81)
50 V=SQR (2*D*9.81)
60 T=INT (T*100)/100
70 V=INT (V*100)/100
80 PRINT "■■TIME TAKEN TO REACH
   GROUND:":PRINT T"SECONDS"
90 PRINT "■■ MAXIMUM VELOCITY
   REACHED :":PRINT V"METRES PER
   SECOND"
100 PRINT "■("INT (2.25*V+.5)
   "MPH )"
200 PRINT "■PRESS ANY KEY TO RUN
   AGAIN"
210 POKE 198,0:WAIT 198,1:
   POKE 198,0
220 RUN
```

```
10 PRINT "■"
20 PRINT "ENTER FALLING DISTANCE
   (METRES)":INPUT D
30 IF D<0 THEN 20
40 T=SQR ((2*D)/9.81)
50 V=SQR (2*D*9.81)
60 T=INT (T*100)/100
70 V=INT (V*100)/100
80 PRINT "■■TIME TAKEN TO REACH
   □□□GROUND□□:":PRINT
   T"SECONDS"
90 PRINT "■■MAXIMUM VELOCITY
   □□□□□□REACHED□:":PRINT
   V"METRES":PRINT "□PER SECOND"
100 PRINT "■("INT (2.25*V+.5)
   "MPH )"
200 PRINT "■■■■■PRESS
   ANY KEY":PRINT "■■■■TO RUN
   AGAIN"
210 POKE 198,0:WAIT 198,1:
   POKE 198,0
220 RUN
```

```
5 @%=&Ø2Ø2ØA
1Ø MODE1
2Ø INPUT"'ENTER THE FALLING DISTANCE IN
   METRES□",D
3Ø IF D<Ø THEN 2Ø
4Ø T=(D*2/9.81)∧.5
5Ø V=(D*2*9.81)∧.5
6Ø PRINT"'TIME TAKEN TO HIT THE GROUND
   :";T;"□SECONDS"
7Ø PRINT"'FINAL VELOCITY:"
   ';V;"□METRES PER SECOND"
8Ø PRINT"(";INT(2.25*V+.5);
   " MPH)"
9Ø PRINT"'PRESS ANY KEY TO RUN
   AGAIN"
1ØØ G$=GET$
11Ø GOTO 1Ø
```

```
1Ø CLS
2Ø INPUT"ENTER FALLING DISTANCE
   (METRES)";D
3Ø IF D<Ø THEN 1Ø
4Ø T=SQR((2*D)/9.81)
5Ø V=SQR(2*D*9.81)
6Ø T=INT(T*1ØØ)/1ØØ
7Ø V=INT(V*1ØØ)/1ØØ
8Ø PRINT@97,"time taken to reach
   ground:":PRINTT;"SECONDS"
9Ø PRINT@225,"maximum velocity
   reached:":PRINTV;"METRES PER SECOND"
1ØØ PRINT"(";INT(2.25*V+.5);
   "MPH)"
11Ø PRINT:PRINT"PRESS ANY KEY TO RUN
   AGAIN"
12Ø IF INKEY$="" THEN 12Ø
13Ø GOTO 1Ø
```

Each computer except the Acorn begins by clearing the screen. The Acorn program starts by formatting the computer (in Line 5). This takes the form of setting the variable @% equal to '&Ø2Ø2ØA'. This tells the computer not to PRINT any number to more (or less) than two decimal places, and set the correct field width. The computer is then put into MODE 1.

After this, the computers all ask you to INPUT the height from which you (or an object) is to fall. If you type in a negative number the computer sends you back to type in another, since you cannot calculate a square root of a negative number.

The computer then calculates the time you or your object would take to reach the ground, and the speed you would be travelling at when you hit it. This is done in Lines 4Ø and 5Ø.

The equation used in Line 4Ø is one version of one of the 'equations of motion'. It has been rearranged so that it takes the following form:

Q+A

Why do I get an error message when I try to work out the square root of a negative number?

Computers cannot calculate the square root of a negative number and, in fact, there can never be such a thing in the real world—although an imaginary value is assigned for some purposes.

When you square a number you multiply it by itself. Positive numbers multiplied together give a positive result, but so do negative numbers. There is no real number that can be squared to give a negative result. The computer naturally gives an error report when asked to do the impossible.

If you want to use the SQR function in a program, then you should make sure that the numbers are always positive. If there is any danger of a negative number arising—perhaps from the result of an earlier calculation or from a number INPUTted into the program—then you should use the ABS function. This converts a number into its absolute value which means that it simply removes any minus sign if there is one.

You can use it in a program by replacing SQR(A) by SQR(ABS(A)), where A represents the number that you're using.

$$T=\sqrt{(2^*D)/a}$$

where **T** is the time taken to travel a given distance (to reach the ground in this case), **D** is the distance, which you INPUT, and **a** is acceleration which measures how much faster you travel every second.

There is no variable **a** in the computer program, since its value does not change, and so you can see 9.81 where **a** would otherwise be.

The equation is solved (or worked out) in Line 4Ø. The answer is the time the object would take to hit the ground.

The next line solves a similar equation to work out what speed it would be travelling at when it hits the ground.

$$V=\sqrt{2^*D^*9.81}$$

In this equation, instead of dividing the number 2*D by the acceleration, the computer multiplies the number by it. V stands for speed (velocity).

In both of the equations, the $\sqrt{\ }$ mark over the '2*D*9.81' or the '2*D/9.81' means 'the square root of'. So that is where you use the square root function in this example. Once you have the answer to either equation, you could change them slightly to get the computer to work out the height from which an object was dropped.

In its general form, the equation applies to any accelerating object from a falling brick to a rocket fired from the moon. What it needs to relate it specifically to the falling object is the right value for the acceleration. In this case, the acceleration is due to gravity, so **a** is set to this value. Gravity has an acceleration of 9.81 metres per second per second. In other words for every second something falls, its speed increases by 9.81 metres per second. Metres per second per second can also be written as metres per second *squared*, so you can see where the power function starts to come in.

The equations to do this would look like this:

$$D=\frac{a^*(t\uparrow 2)}{2}$$

or

$$D=V\uparrow 2/(a^*2)$$

Try to change the computer program to work out the answers using these two equations. Instead of the height, you could INPUT the speed at which you would like something to hit the ground (for the second equation) or the time that you would like it to take before reaching the ground. In both cases your computer would work out how high you would need to be to achieve your aims.

The ability to solve problems of this kind has all sorts of practical uses—although the equations usually have to take into account other influences on the moving body as well. But you could, for example, work out the details of a car's performance in acceleration and braking tests—or reconstruct what happened during an accident. In the latter case, you might want to know what speed the vehicle was travelling when it crashed. It is possible to do this working back from the distance it travelled after impact, as long as you have the right facts to put into the equation.

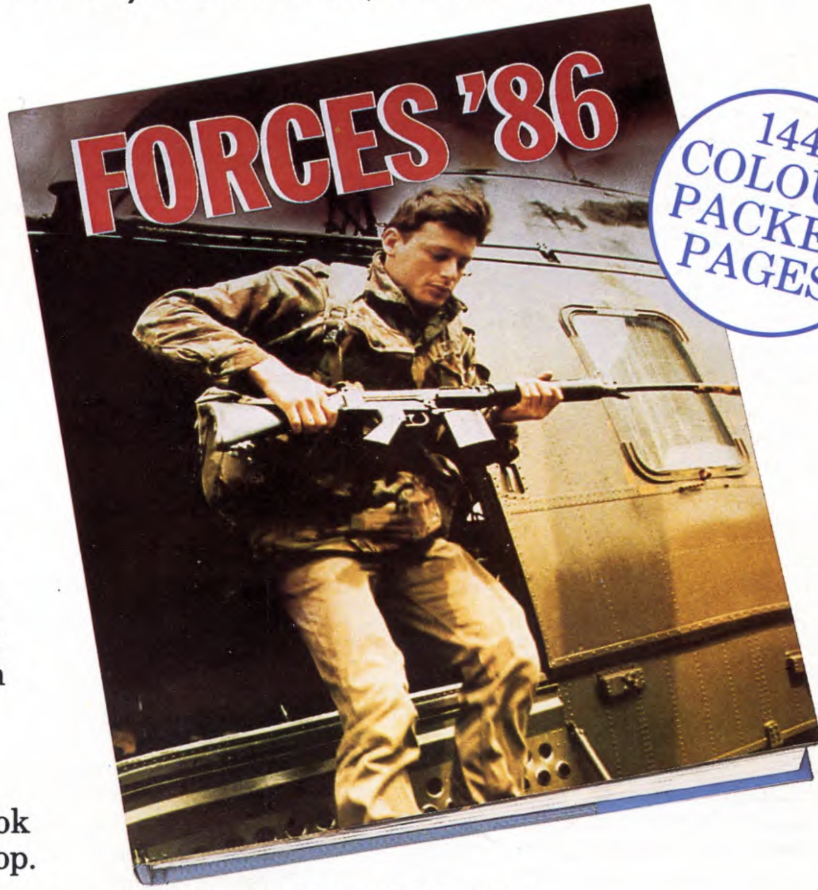
It isn't just things like the size of your house or the performance of your car that can be modelled by using the power functions. They also apply to things as diverse as the way a tree grows and why a bird as big as an elephant wouldn't be able to get off the ground! In a later article, you will see how to make even more use of the mathematics functions on your computer, including how to achieve some striking graphics effects.

REVEALED *the dramas and tensions, rumours and half truths – in war and peace.*

FIND OUT *what it means to have to fight, the price of keeping the peace.*
what faces British Military Forces in '86, and the future.

FORCES '86

A fascinating and revealing insight into the who, what and why driving British Military Forces into 1986, and beyond.



144 COLOUR PACKED PAGES

FORCES '86

Looks at Britain's military scene as a whole. Personal accounts of marine training, combat flying and the submarine service compliment articles on war in space, a major European exercise and a historic and unique Regiment to present a stimulating and informative view of British armed forces today.

FORCES '86

Is an important and engrossing book that is not available in any book shop.

From the publishers of THE FALKLANDS WAR

SPECIAL OFFER ORDER FORM

FORCES '86

Simply fill in your name and address, cut out the coupon, put it in an envelope and post today.
No stamp is needed.

Address your envelope to:

Marshall Cavendish,
FREEPOST,
Lambourn Woodlands,
Hungerford, Berks. RG16 7BR

☐ **YES!** Please send me Forces '86, on approval, with an invoice for £7.95 (plus 95p postage and packing).

Name______________________________

Address______________________________

____

Tel. No.______________________________

Signature________________________(I am over 18)

GUARANTEE

- Marshall Cavendish guarantees that if you are not entirely satisfied with your book and return it within 14 days, you will owe nothing.
- As a subscriber to Forces '86 you will be advised of, and entitled to receive on approval, any further annual publications.

F86/IP

Registered London 385817 BF12



Coming soon in INPUT



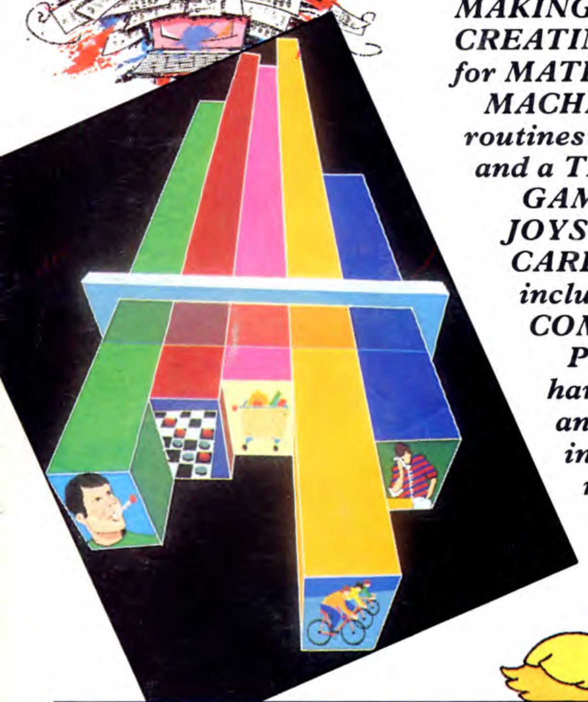
VOLUME TWO

Explore more of your computer's BASIC facilities. In GRAPHICS, there's WIREFRAME DRAWING and new ideas for UDGs. And in SOUND, get started on MUSIC MAKING. There's information on CREATING FILES and new uses for MATHS, plus lots more . . .

MACHINE CODE gives you useful routines including PACKERS—and a TRACE to help find the bugs.

GAMES shows how to use JOYSTICKS, how to program a CARD GAME, and more—including a practical TEXT COMPRESSOR.

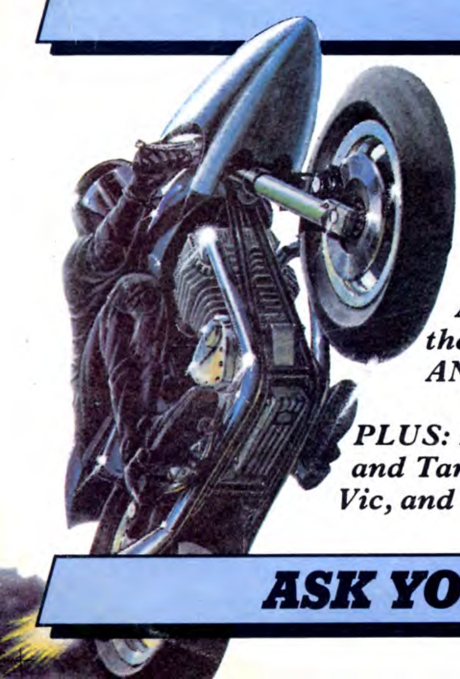
PERIPHERALS looks at new hardware including MODEMS and STORAGE UNITS. And in APPLICATIONS, there is more useful software like a TEXT EDITOR and a DRAWING PROGRAM.







COMING IN ISSUE 14...

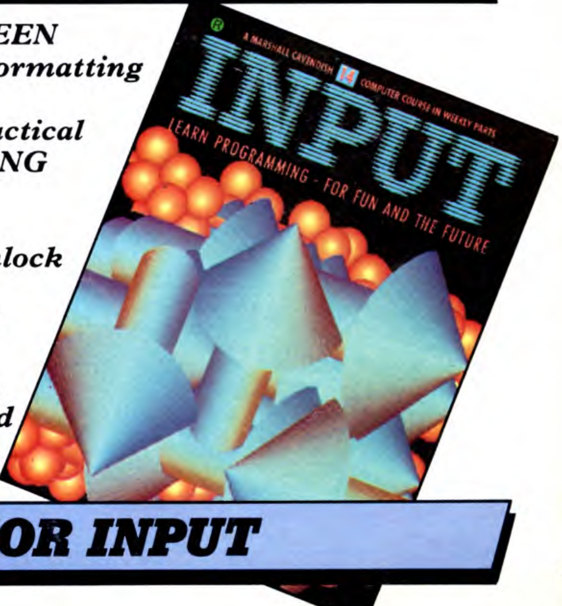


For brighter, neater SCREEN DISPLAYS, find out about formatting

And for displays with a practical purpose, learn about DRAWING LINE GRAPHS

Avid adventurers will learn to unlock the secrets of THE ADVENTURE AND HOW TO PLAN YOUR OWN

PLUS: An ASSEMBLER for the Dragon and Tandy, GRAPHICS for the ZX81 and Vic, and to COMMODORE SYMBOLS



ASK YOUR NEWSAGENT FOR INPUT